



DeepRob

Lecture 10
Training Neural Networks II
University of Michigan and University of Minnesota



Recap

1. One time setup:

- Activation functions, data preprocessing, weight initialization, regularization

Last time

2. Training dynamics:

- Learning rate schedules; large-batch training; hyperparameter optimization

Today

3. After training:

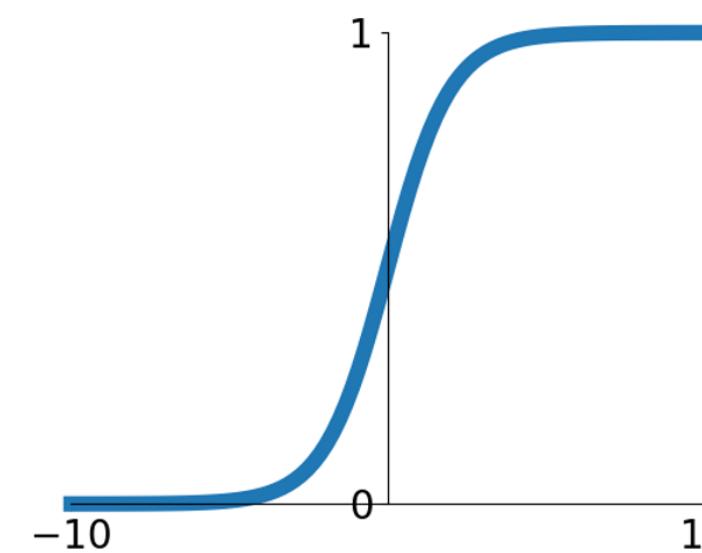
- Model ensembles, transfer learning



Last time: Activation Functions

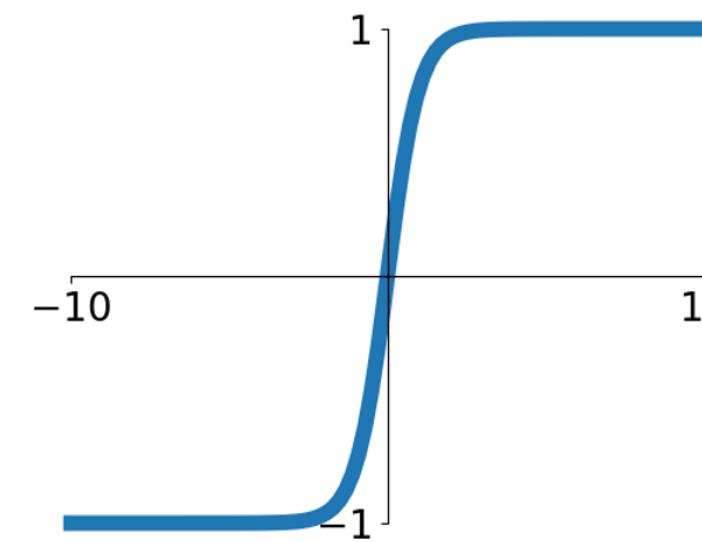
Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



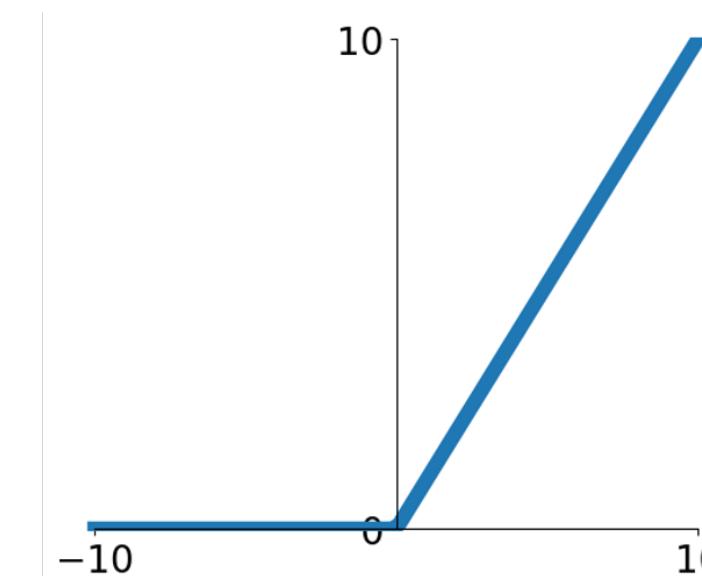
tanh

$$\tanh(x)$$



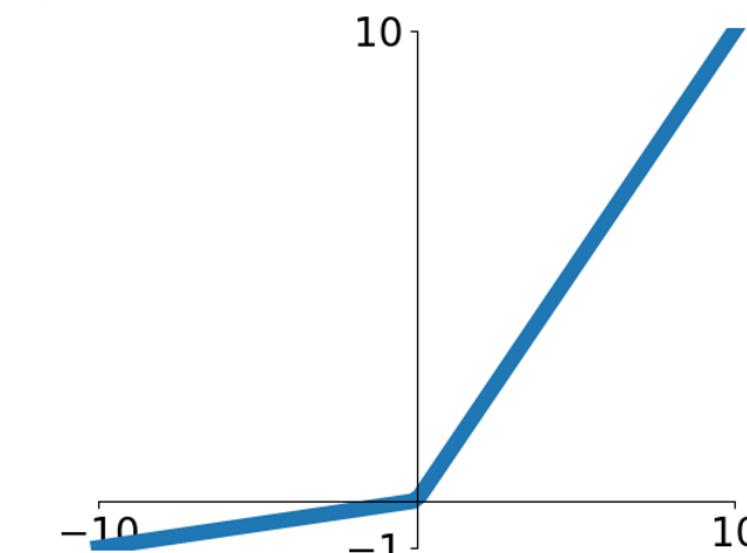
ReLU

$$\max(0, x)$$



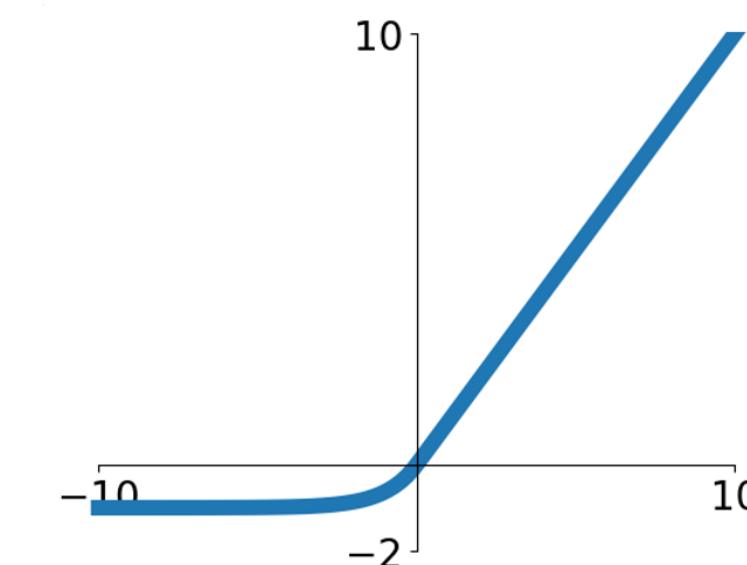
Leaky ReLU

$$\max(0.1x, x)$$



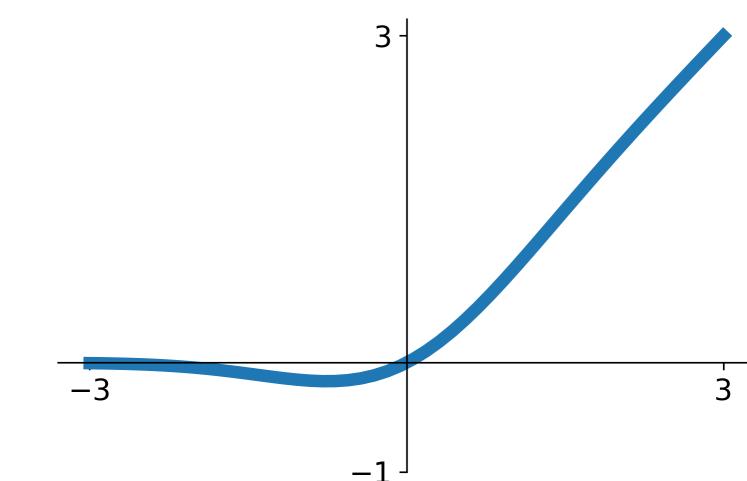
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(\exp^x - 1) & x < 0 \end{cases}$$



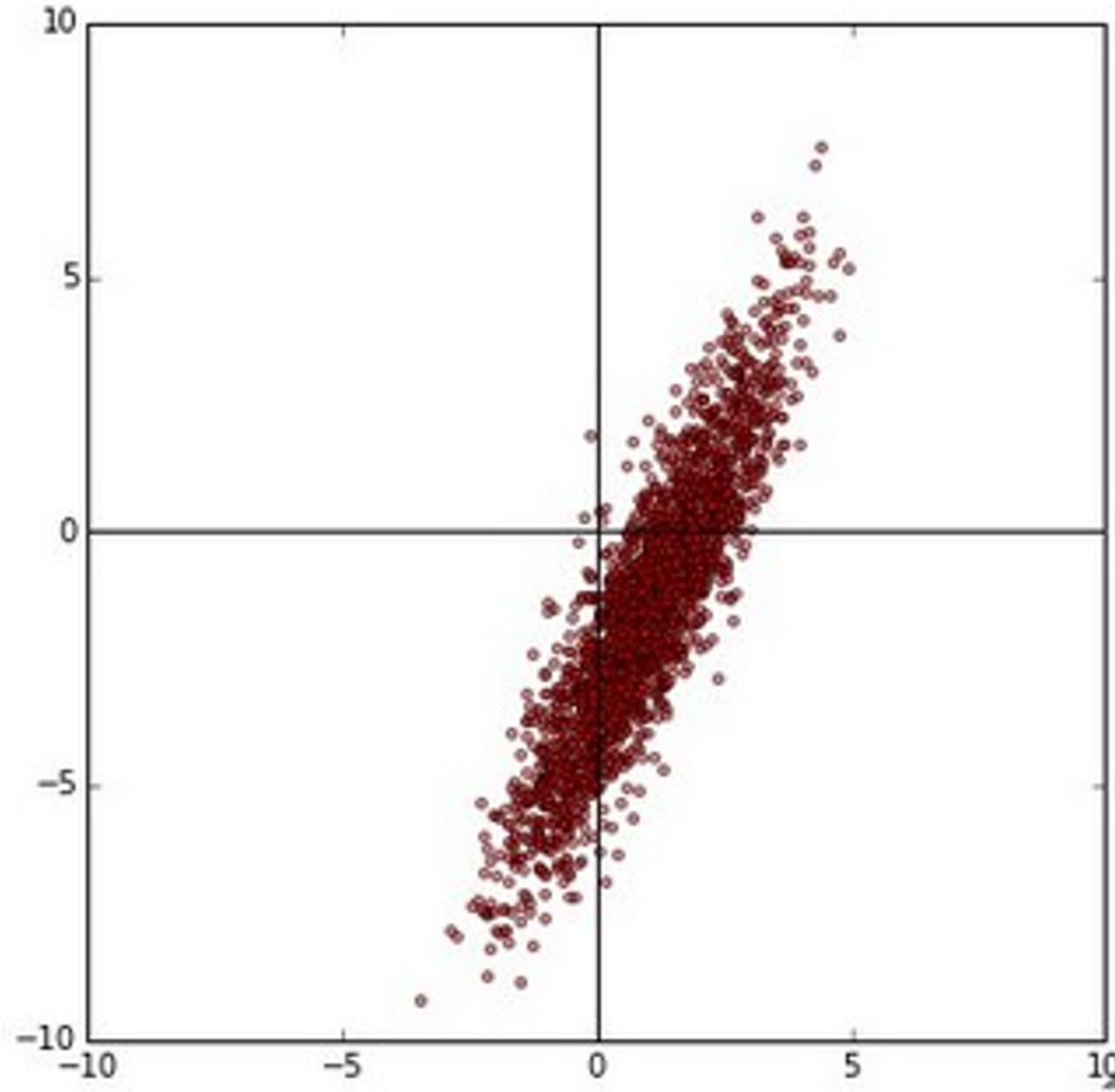
GELU

$$\approx x\alpha(1.702x)$$

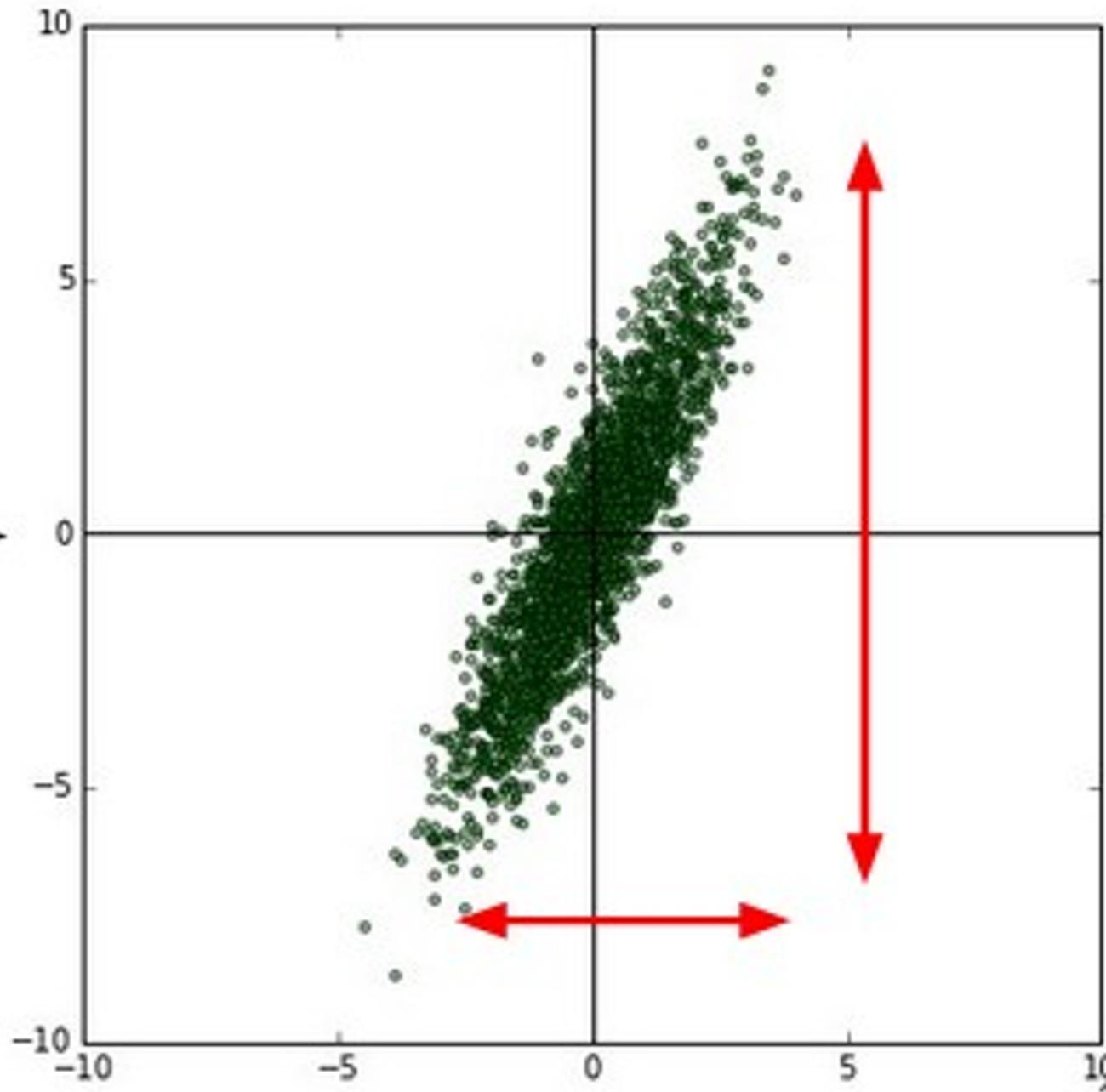


Last time: Data Preprocessing

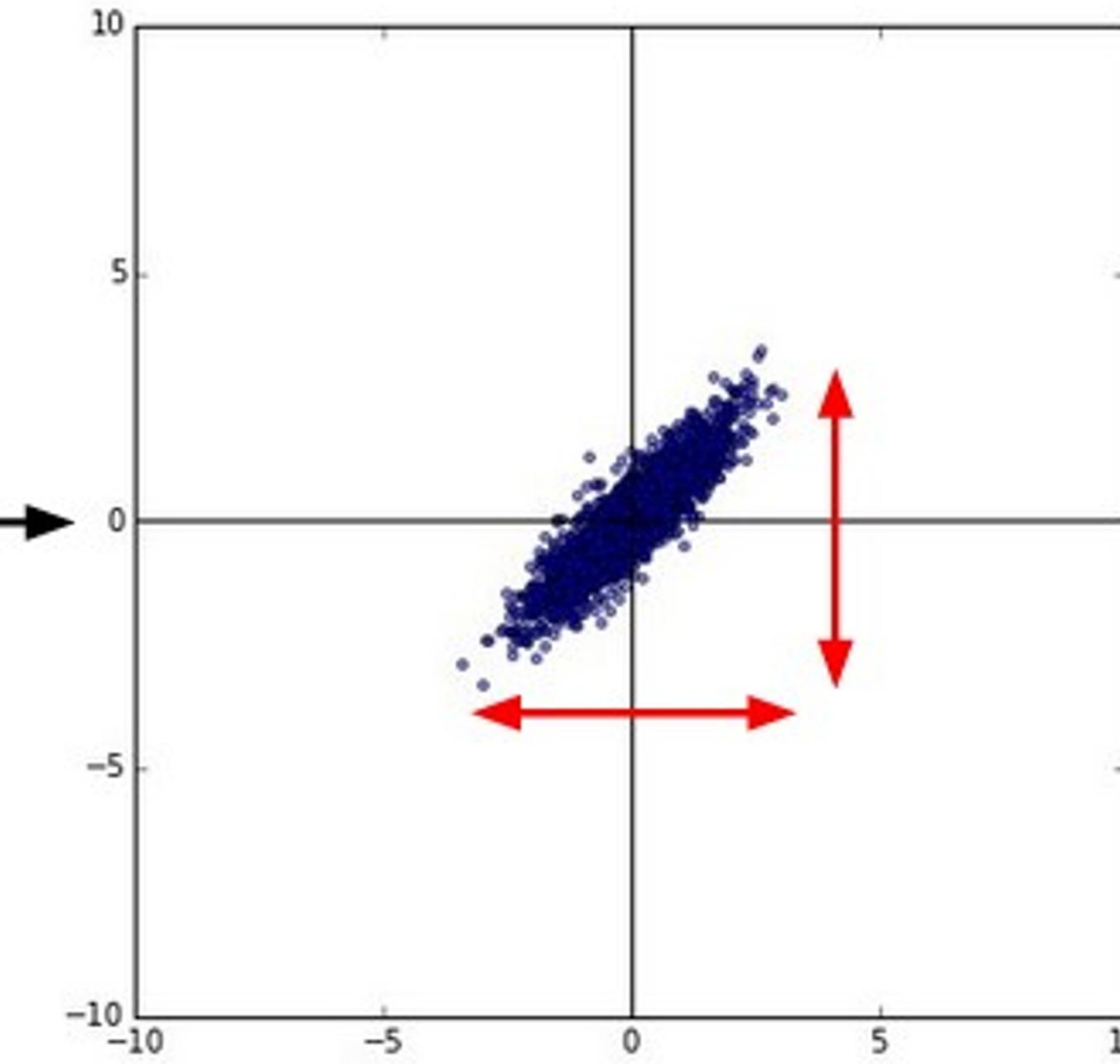
original data



zero-centered data



normalized data



Last time: Weight initialization

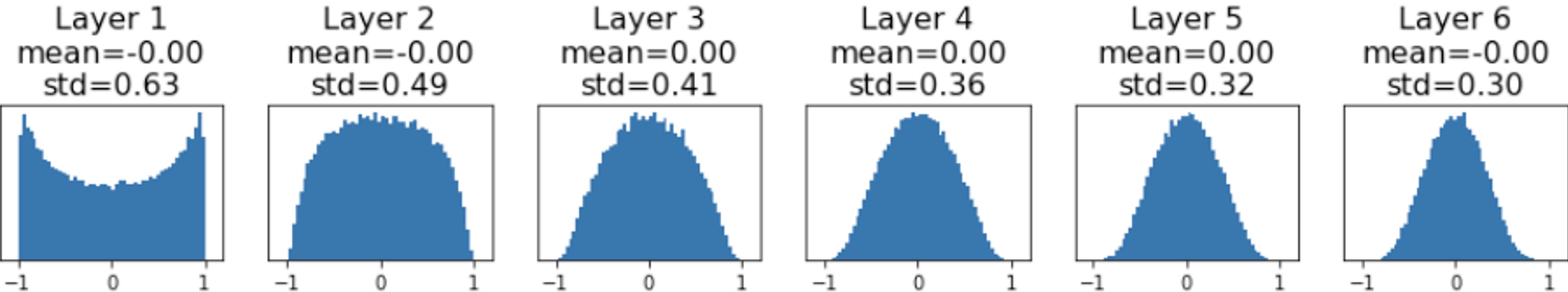
```

dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)

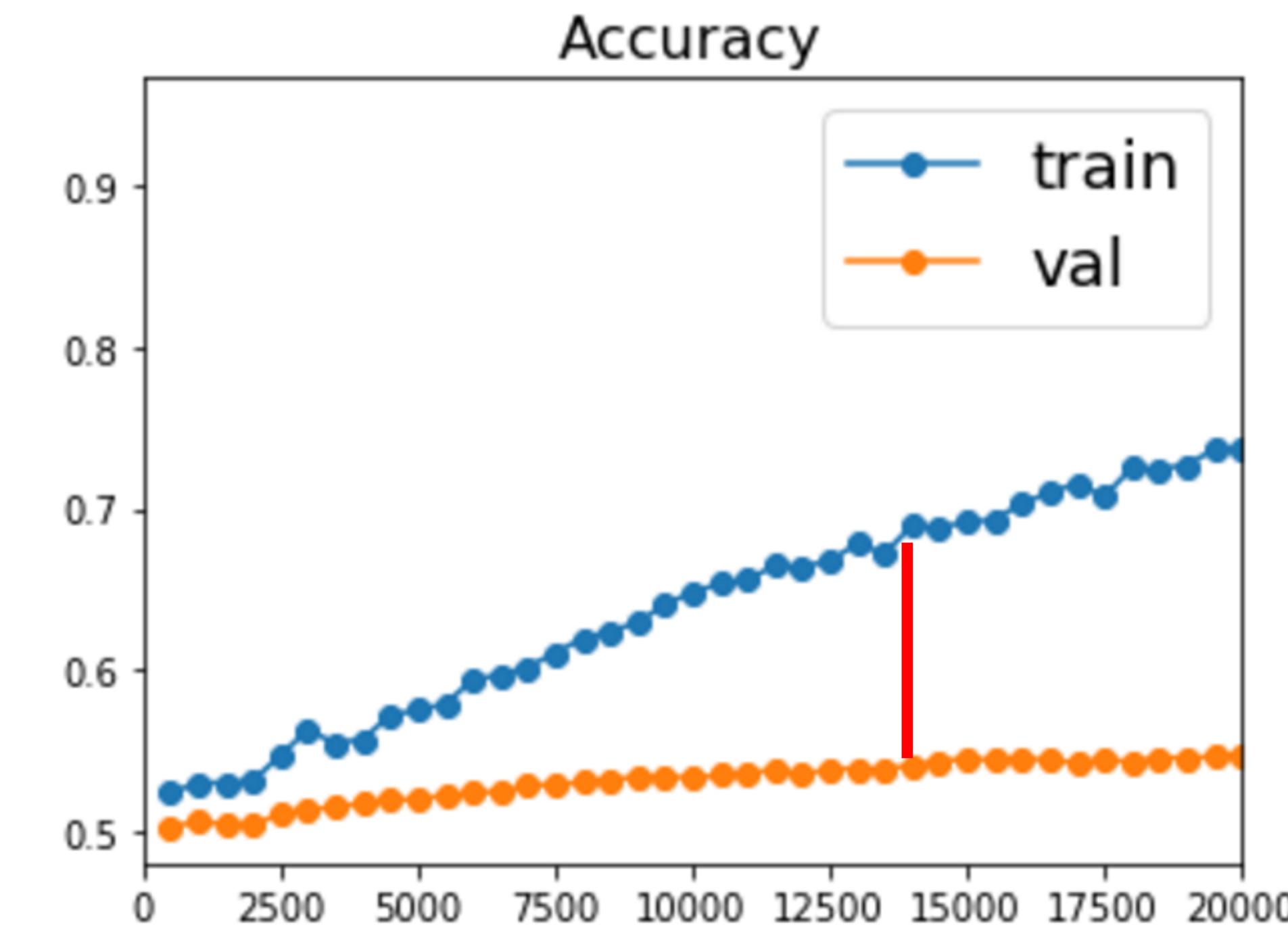
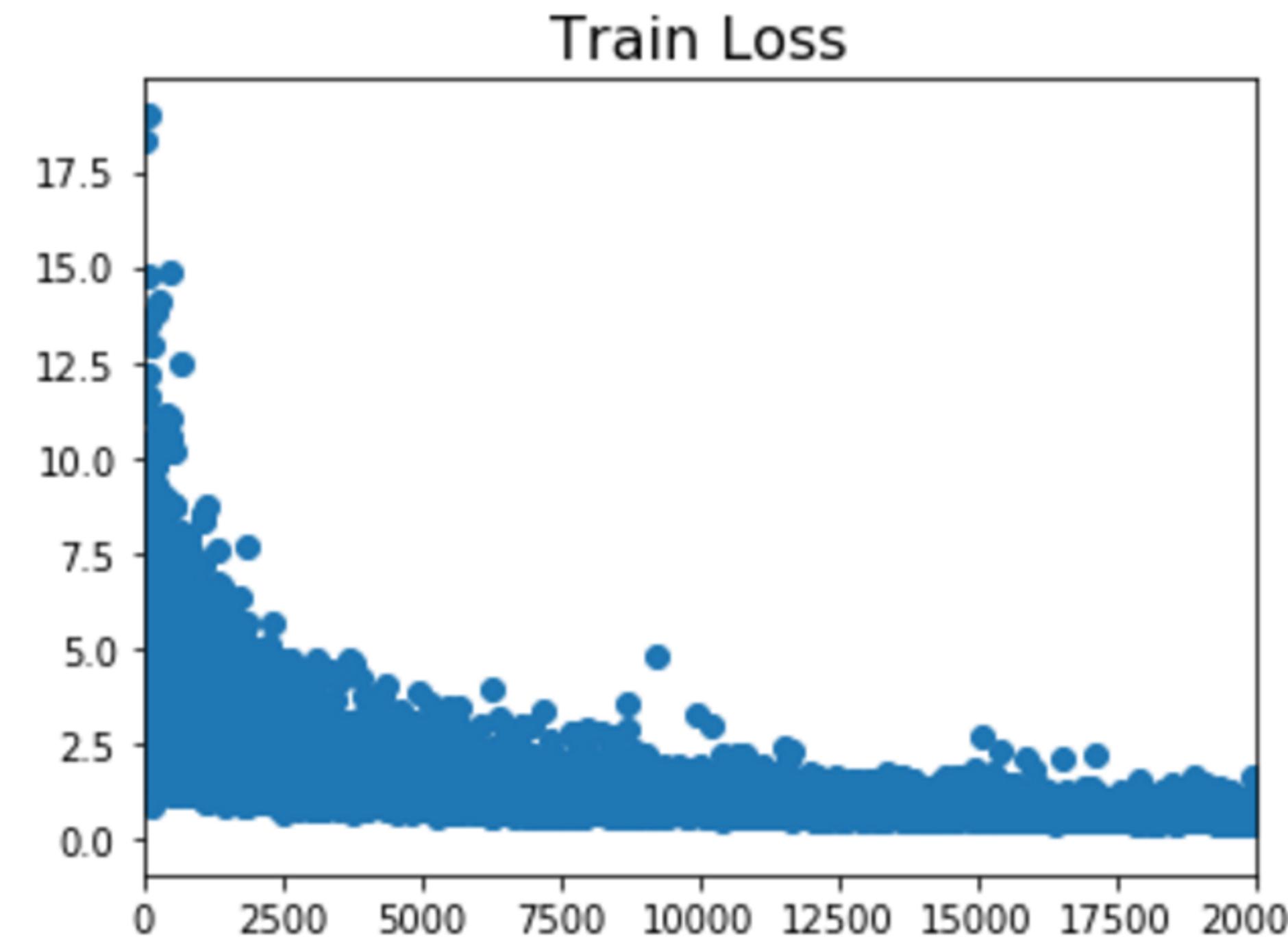
```

“Xavier” initialization:
 $\text{std} = 1/\sqrt{\text{Din}}$

“Just right”: Activations are nicely scaled for all layers!



Now your model is training ... but it overfits!



Regularization



Regularization: Add term to the loss

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W)$$

In common use:

L2 regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2 \text{ (Weight decay)}$$

L1 regularization

$$R(W) = \sum_k \sum_l |W_{k,l}|$$

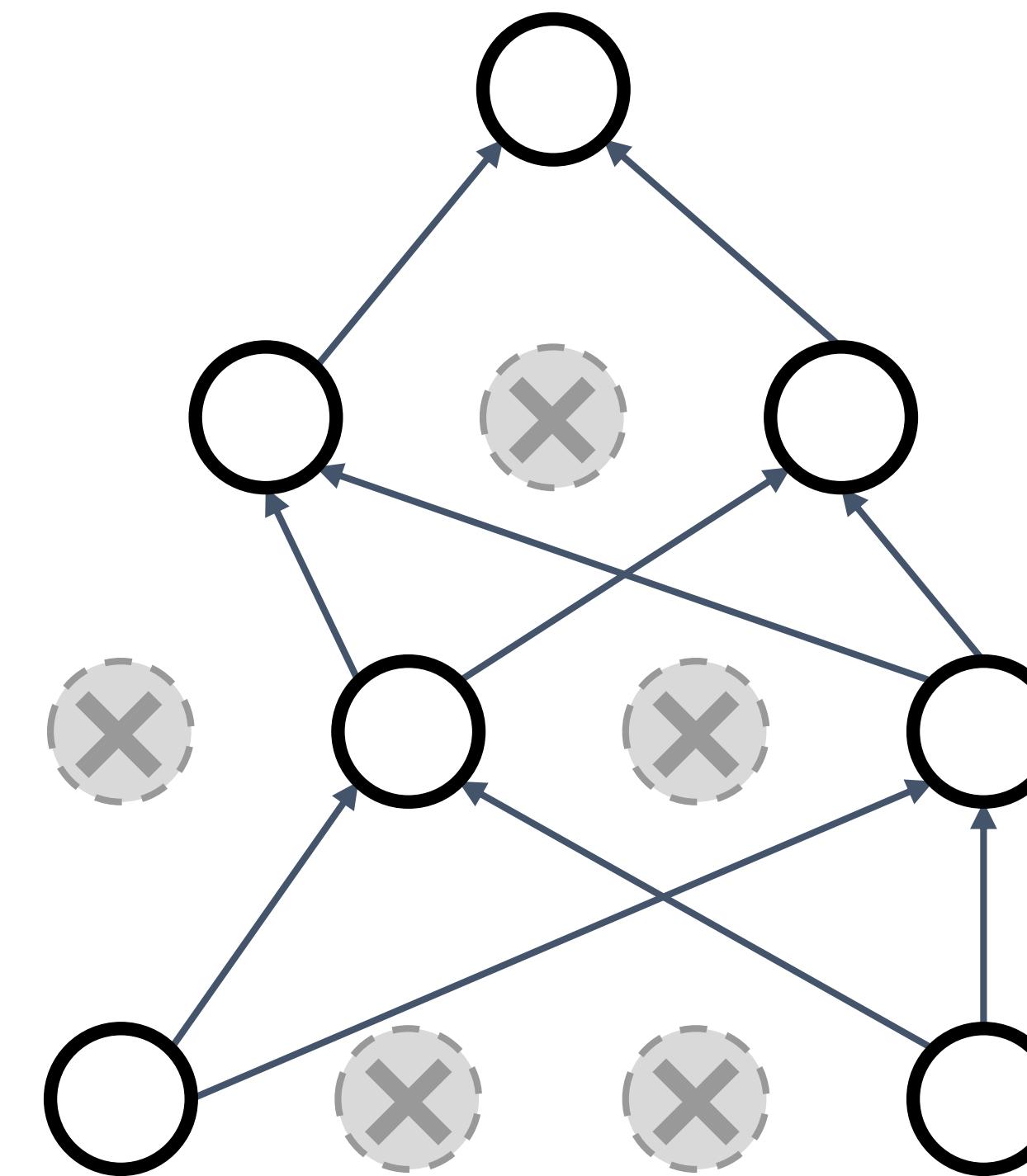
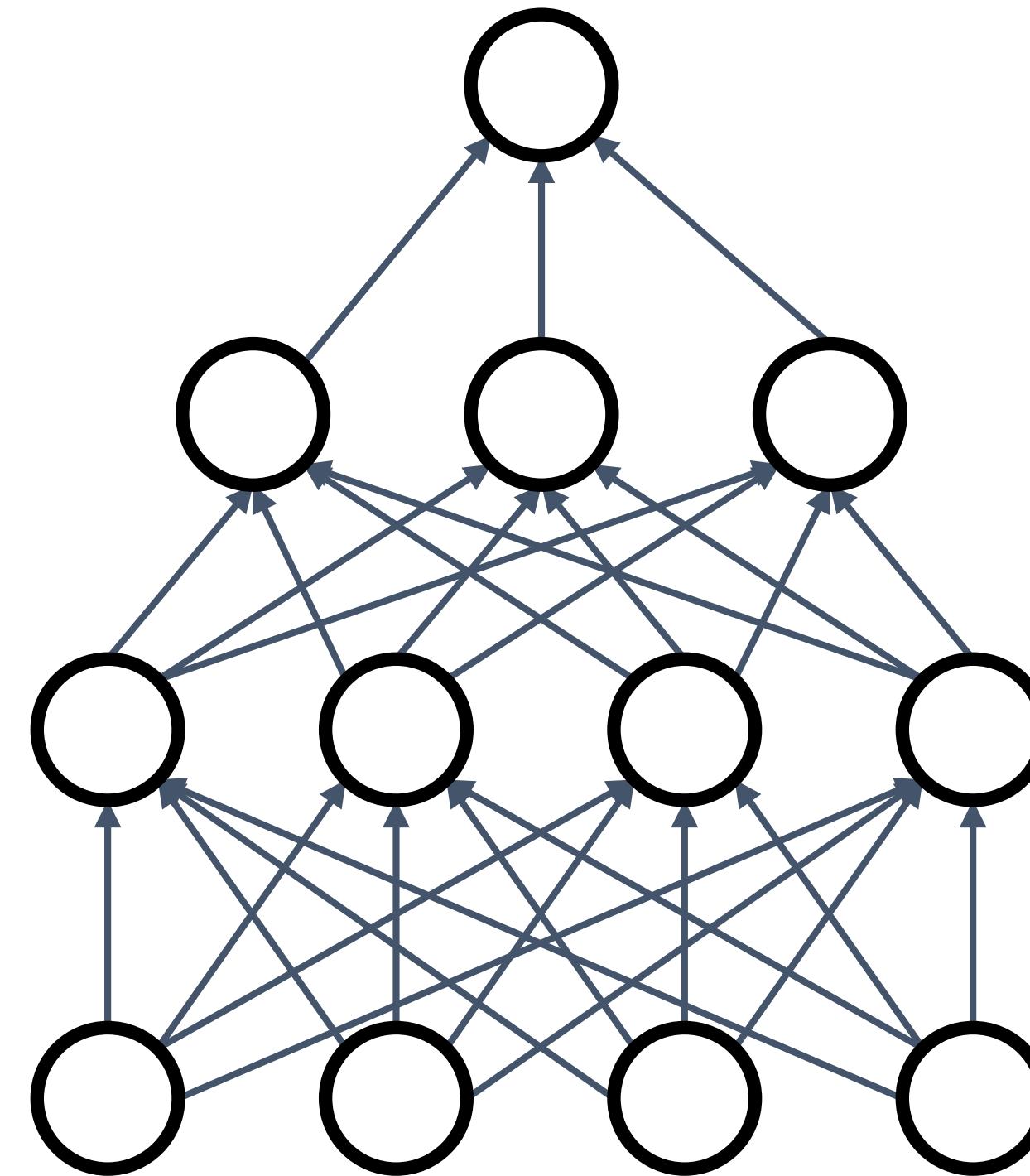
Elastic net (L1 + L2)

$$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$$



Regularization: Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common



Regularization: Dropout

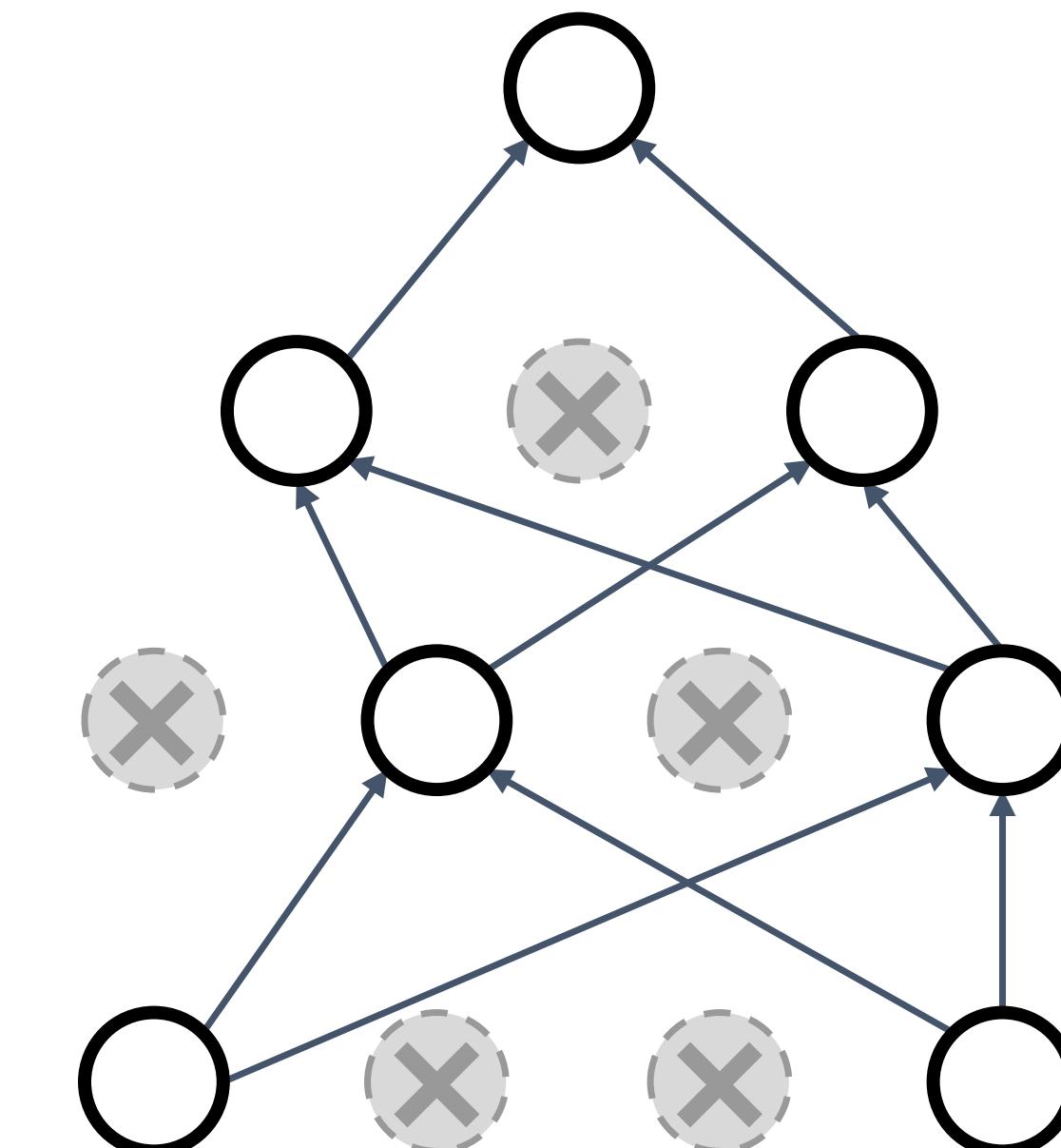
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

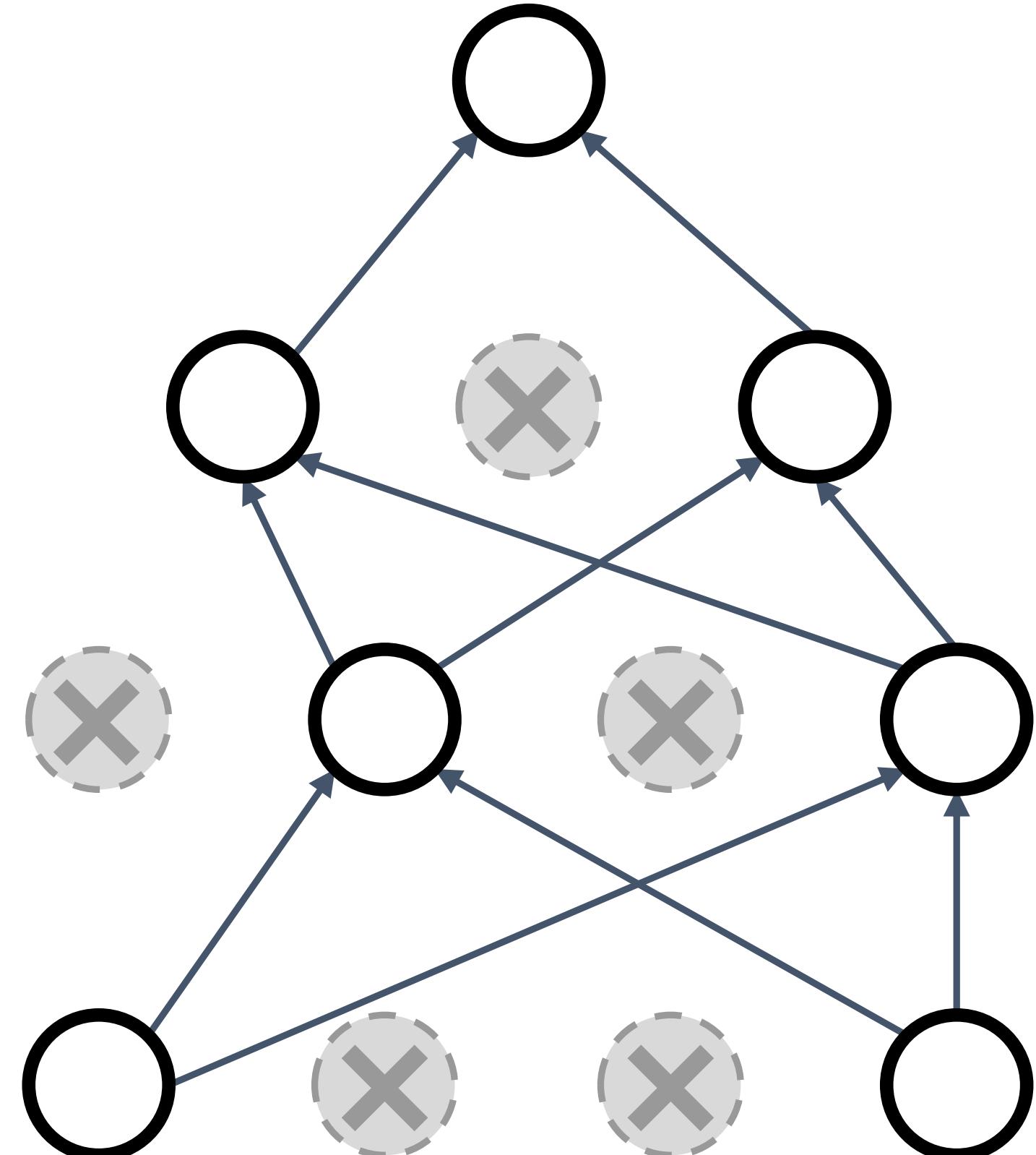
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
```

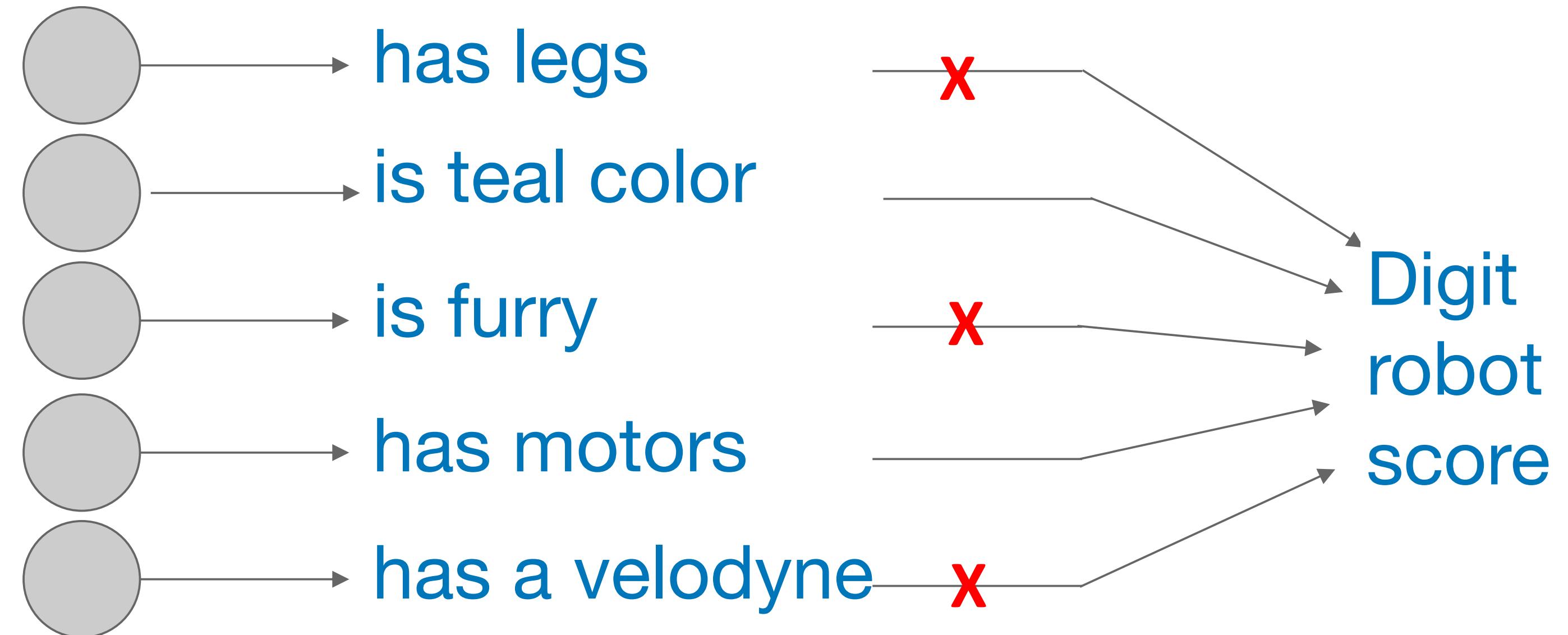
Example forward pass
with a 3-layer network
using dropout



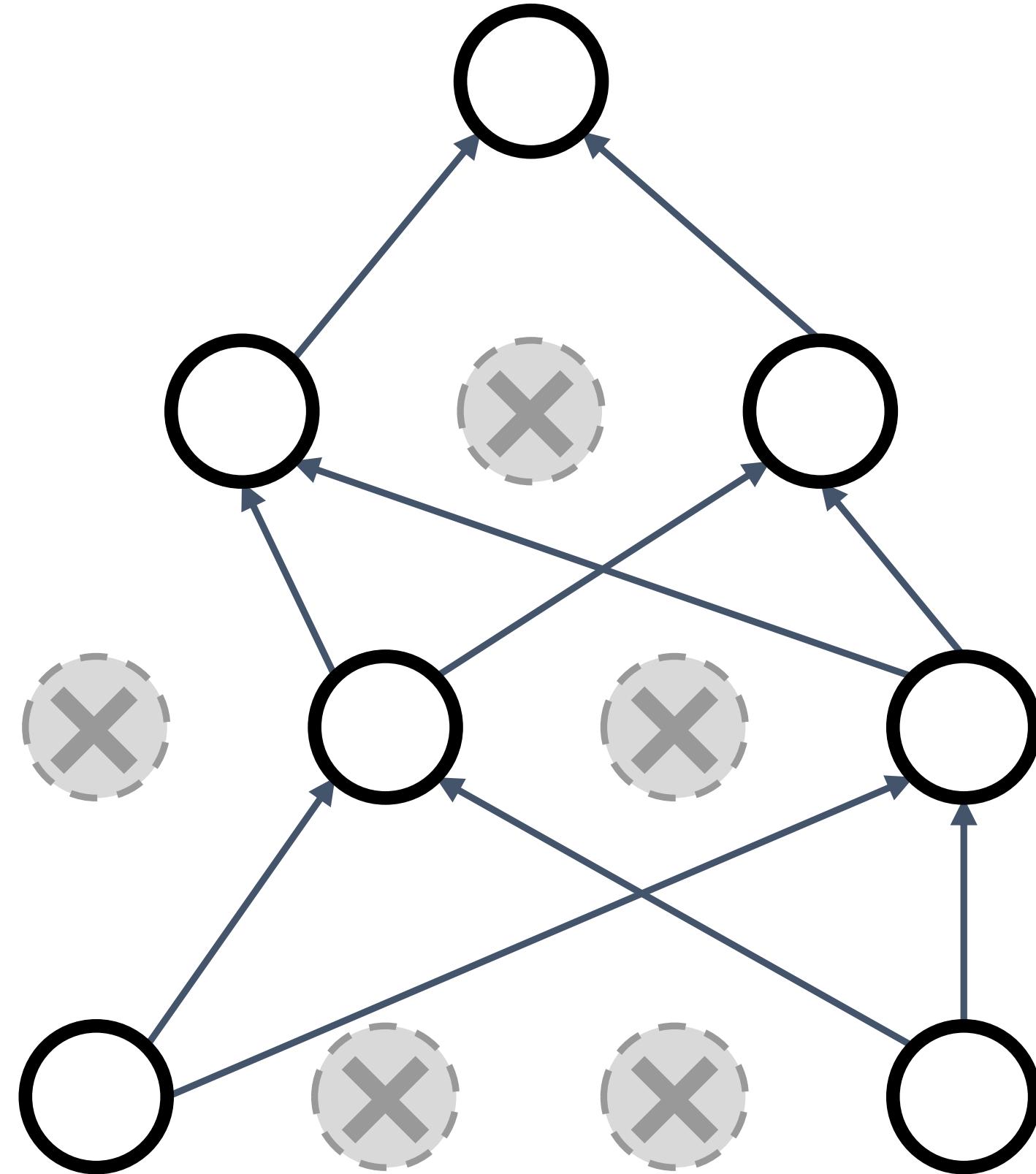
Regularization: Dropout



Forces the network to have a redundant representation; prevents **co-adaptation** of features



Regularization: Dropout



Another interpretation:

Dropout is training a large *ensemble* of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!

Only $\sim 10^{82}$ atoms in the universe...

Dropout: Test time

Dropout makes our output random!

$$y = f_w(x, z)$$

Want to “average out” the randomness at test-time

$$y = f(x, z) = \mathbb{E}_z[f(x, z)] = \int p(z)f(x, z)dz$$

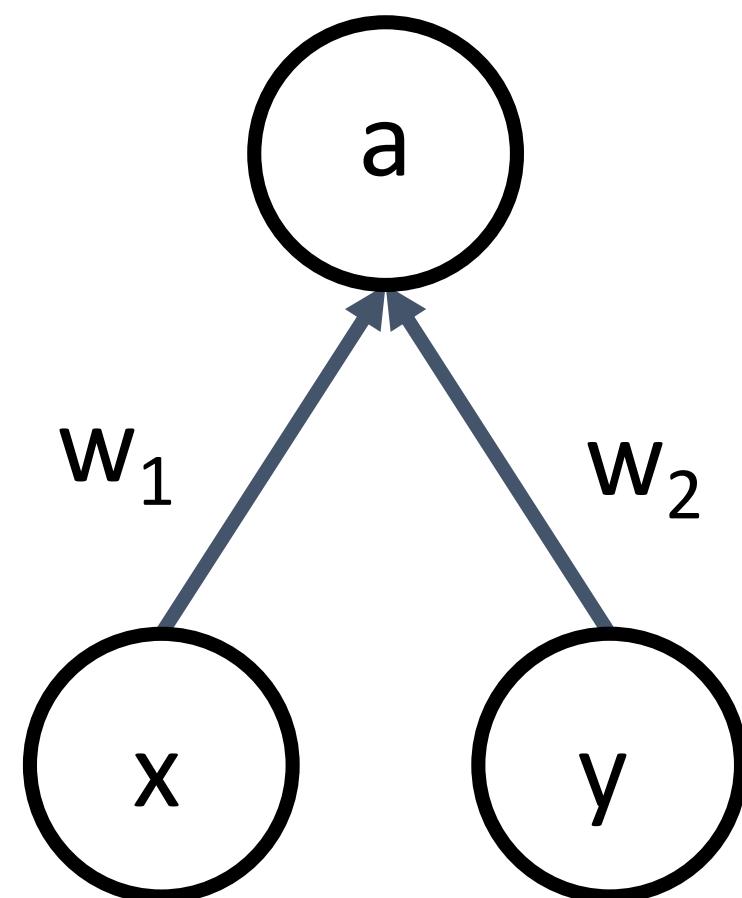
But this integral seems hard...



Dropout: Test time

Want to approximate
the integral

$$y = f(x, z) = \mathbb{E}_z[f(x, z)] = \int p(z)f(x, z)dz$$



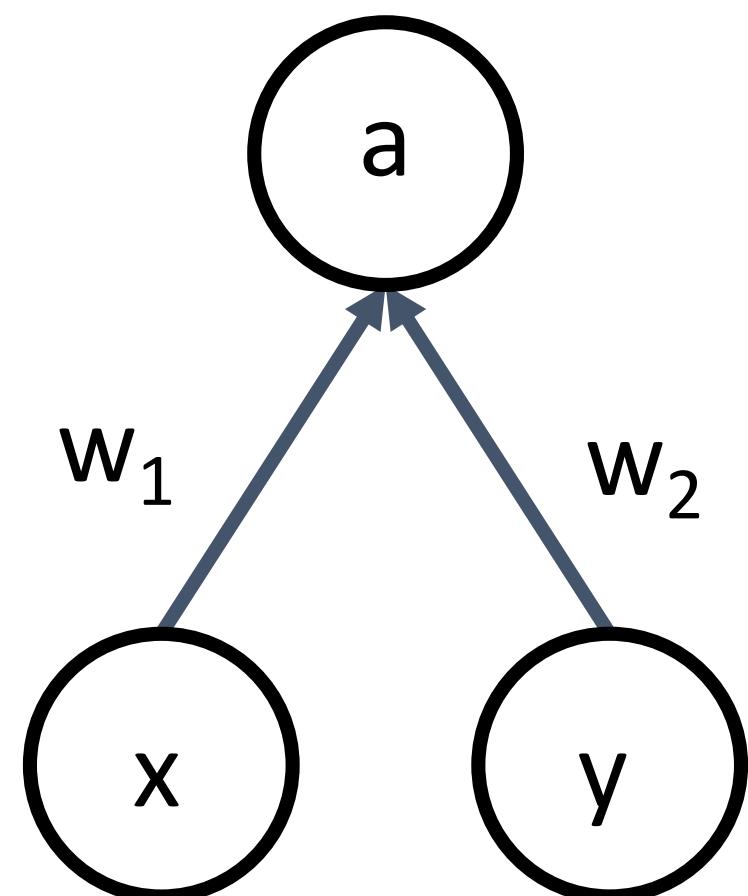
Consider a single neuron:

At test time we have: $\mathbb{E}[a] = w_1x + w_2y$

Dropout: Test time

Want to approximate
the integral

$$y = f(x, z) = \mathbb{E}_z[f(x, z)] = \int p(z)f(x, z)dz$$



Consider a single neuron:

At test time we have: $\mathbb{E}[a] = w_1x + w_2y$

During training time we have: $\mathbb{E}[a] = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y)$

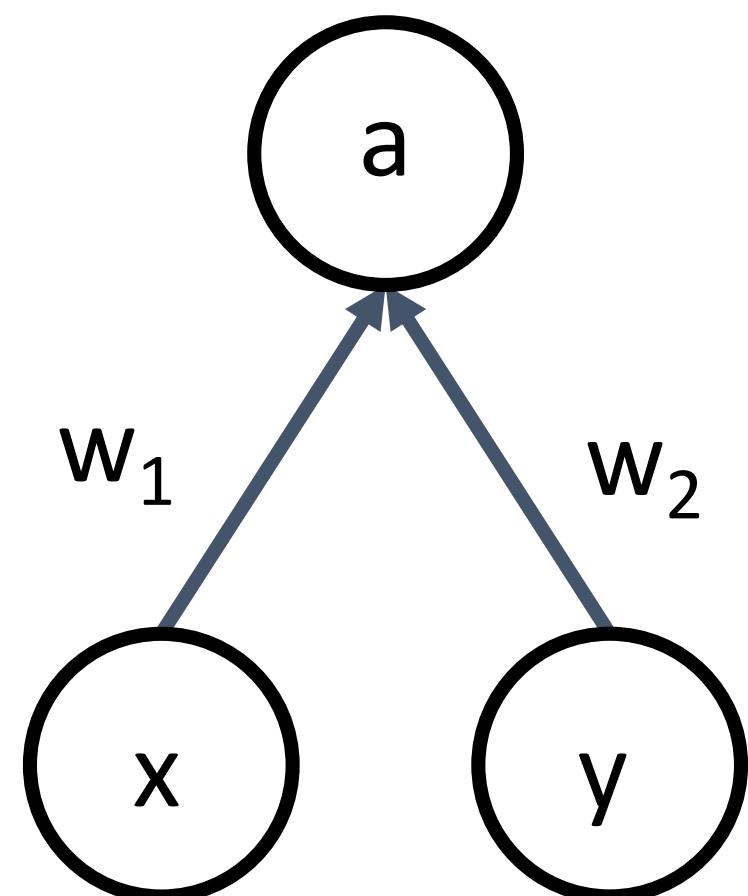
$$+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y)$$

$$= \frac{1}{2}(w_1x + w_2y)$$

Dropout: Test time

Want to approximate
the integral

$$y = f(x, z) = \mathbb{E}_z[f(x, z)] = \int p(z)f(x, z)dz$$



Consider a single neuron:

At test time we have: $\mathbb{E}[a] = w_1x + w_2y$

During training time we have: $\mathbb{E}[a] = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y)$

At test time, drop nothing and *multiply* by dropout probability

$$\begin{aligned}
 &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\
 &= \frac{1}{2}(w_1x + w_2y)
 \end{aligned}$$

Dropout: Test time

```
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always

=> We must scale the activations so that for each neuron:

Output at test time = Expected output at training time

Dropout Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

Drop in forward pass

Scale at test time



More common: “Inverted dropout”

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

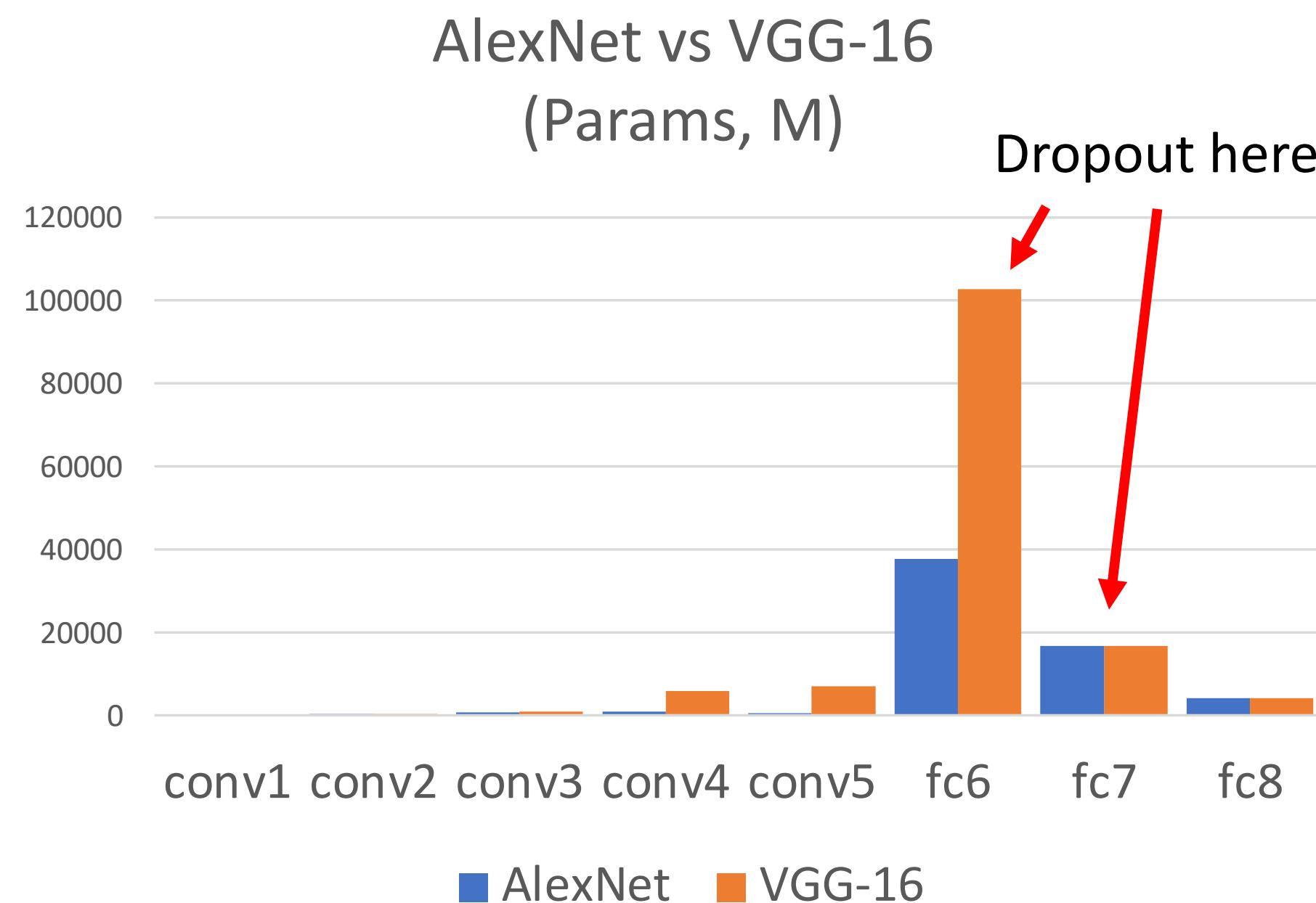
Drop and scale
during training

test time is unchanged!



Dropout architectures

Recall AlexNet, VGG have most of their parameters in **fully-connected layers**; usually Dropout is applied there



Later architectures (GoogLeNet, ResNet, etc) use global average pooling instead of fully-connected layers: they don't use dropout at all!

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_w(x, z)$$

Testing: Average out randomness
(sometimes approximate)

$$y = f(x, z) = \mathbb{E}_z[f(x, z)] = \int p(z)f(x, z)dz$$



Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_w(x, z)$$

For ResNet and later,
often L2 and Batch
Normalization are the
only regularizers!

Testing: Average out randomness
(sometimes approximate)

$$y = f(x, z) = \mathbb{E}_z[f(x, z)] = \int p(z)f(x, z)dz$$

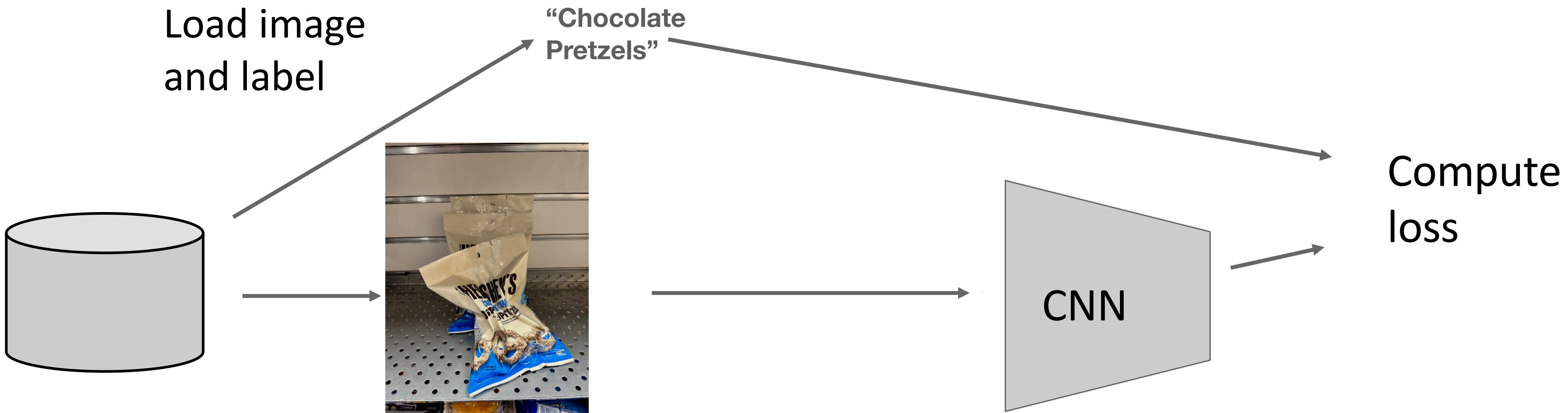
Example: Batch Normalization

Training: Normalize using stats
from random mini batches

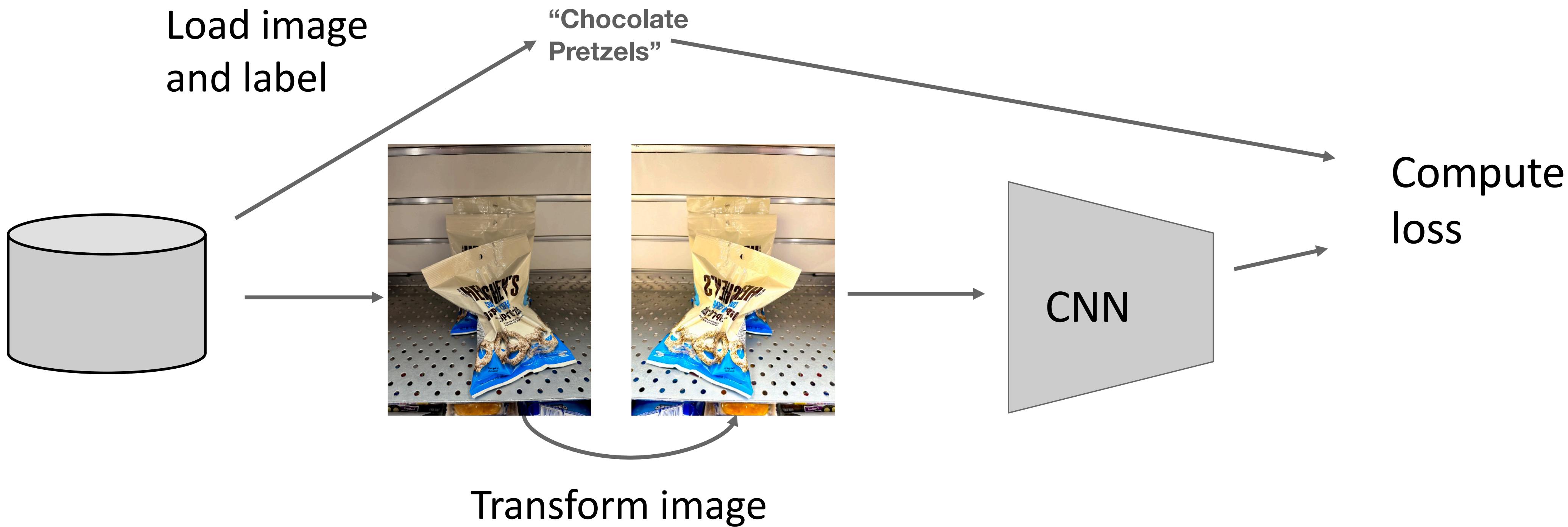
Testing: Use fixed stats to
normalize



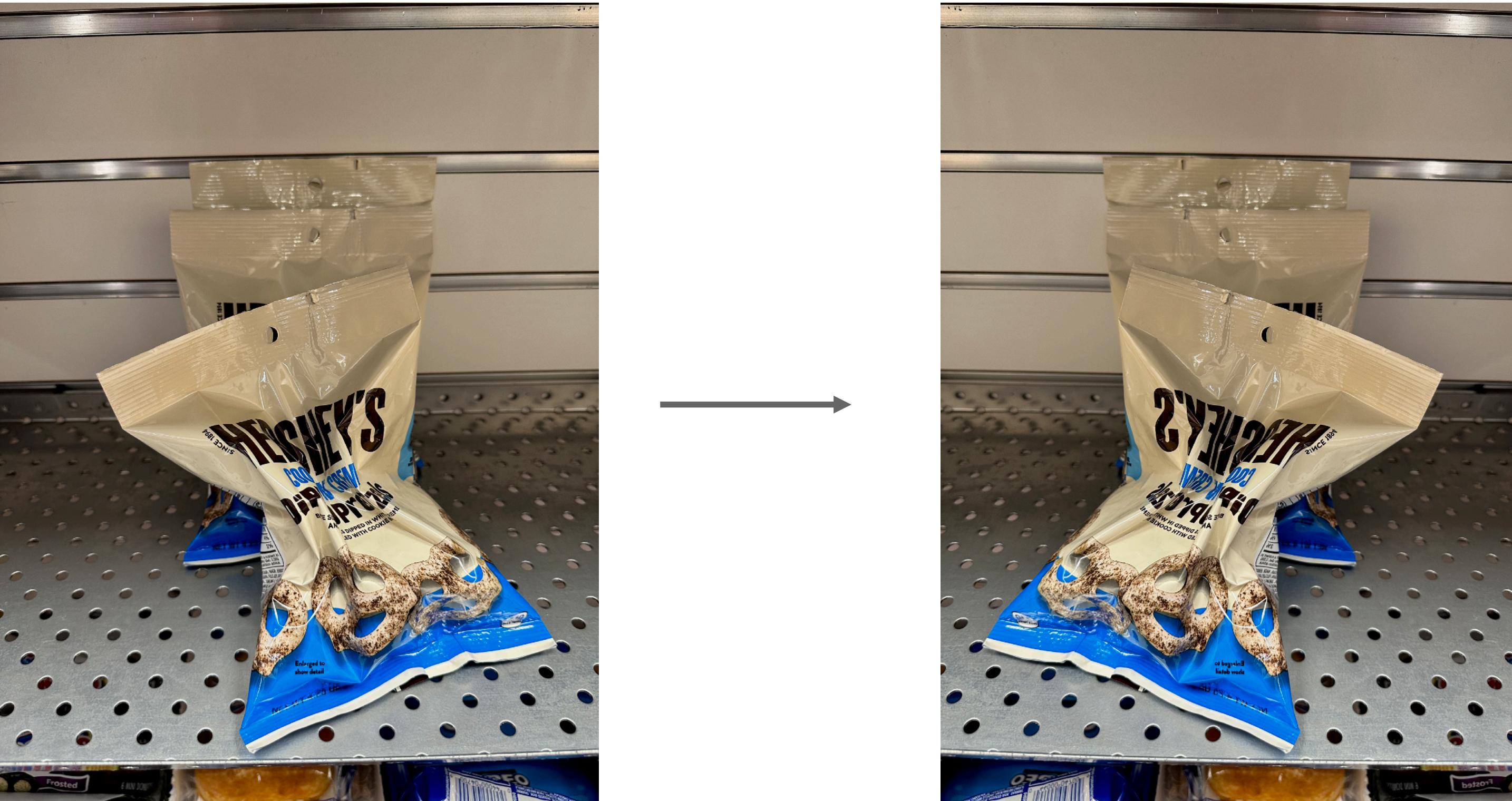
Data augmentation



Last time: Data augmentation



Data augmentation: Horizontal Flips



Data augmentation: Random Crops and Scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224×224 patch

Testing: average a fixed set of crops

ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224×224 crops: 4 corners + center, + flips



Data augmentation: Color Jitter

Simple: Randomize contrast and brightness



More complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(Used in AlexNet, ResNet, etc)

Data augmentation: RandAugment

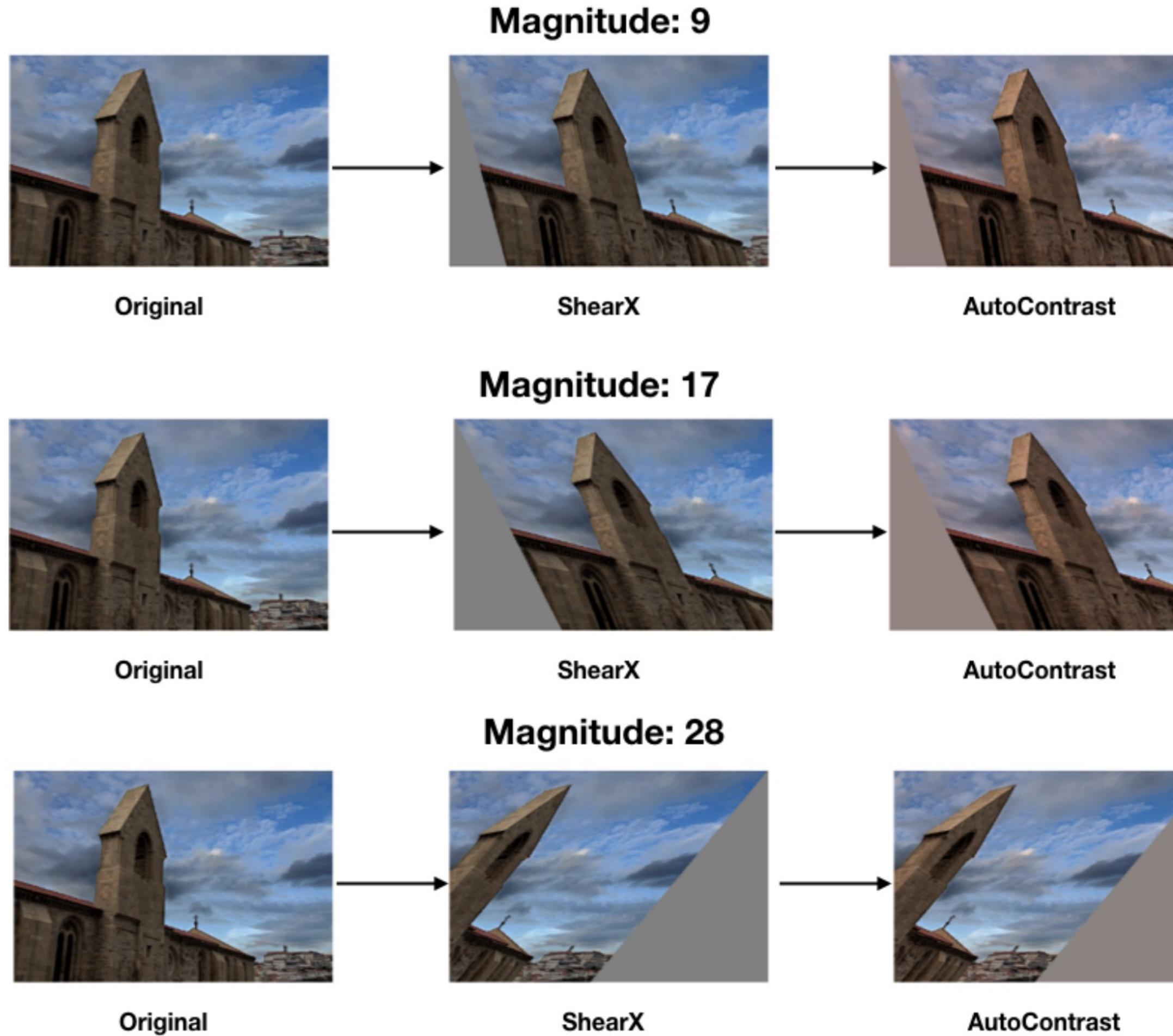
```
transforms = [  
    'Identity', 'AutoContrast', 'Equalize',  
    'Rotate', 'Solarize', 'Color', 'Posterize',  
    'Contrast', 'Brightness', 'Sharpness',  
    'ShearX', 'ShearY', 'TranslateX', 'TranslateY']  
  
def randaugment(N, M):  
    """Generate a set of distortions.  
  
    Args:  
        N: Number of augmentation transformations to  
            apply sequentially.  
        M: Magnitude for all the transformations.  
    """  
  
    sampled_ops = np.random.choice(transforms, N)  
    return [(op, M) for op in sampled_ops]
```

Apply random combinations of transforms:

- **Geometric:** Rotate, translate, shear
- **Color:** Sharpen, contrast, brightness, solarize, posterize, color



Data augmentation: RandAugment



**Apply random combinations
of transforms:**

- **Geometric:** Rotate, translate, shear
- **Color:** Sharpen, contrast, brightness, solarize, posterize, color

Data augmentation: Get creative for your problem!

Data augmentation encodes **invariances** in your model

Think for your problem: what changes to the image should **not** change the network output?

Maybe different for different tasks!

Regularization: A common pattern

Training: Add some randomness

Testing: Marginalize over randomness

Examples:

Dropout

Batch Normalization

Data Augmentation



Regularization: DropConnect

Training: Drop random connections between neurons (set weight=0)

Testing: Use all the connections

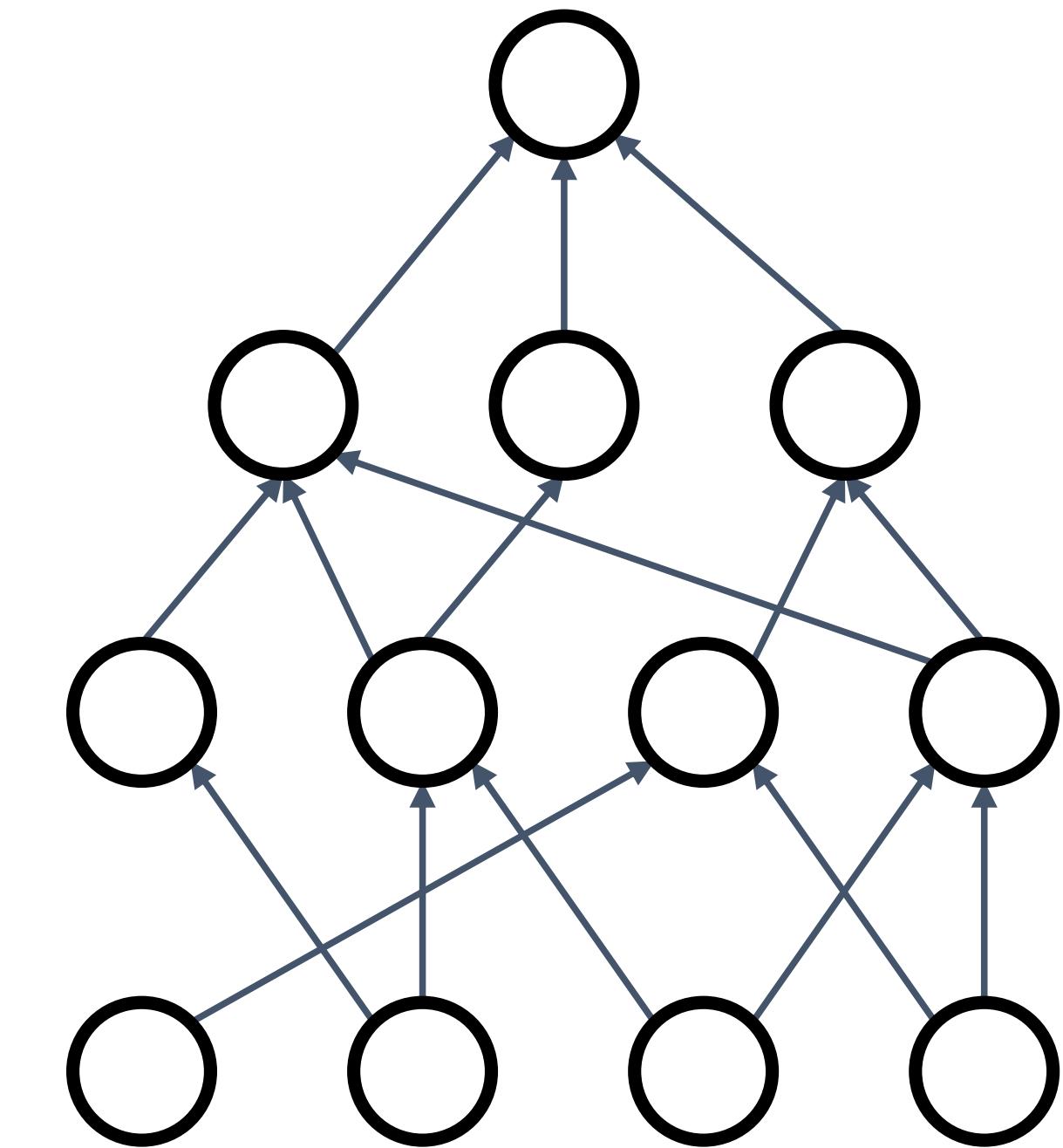
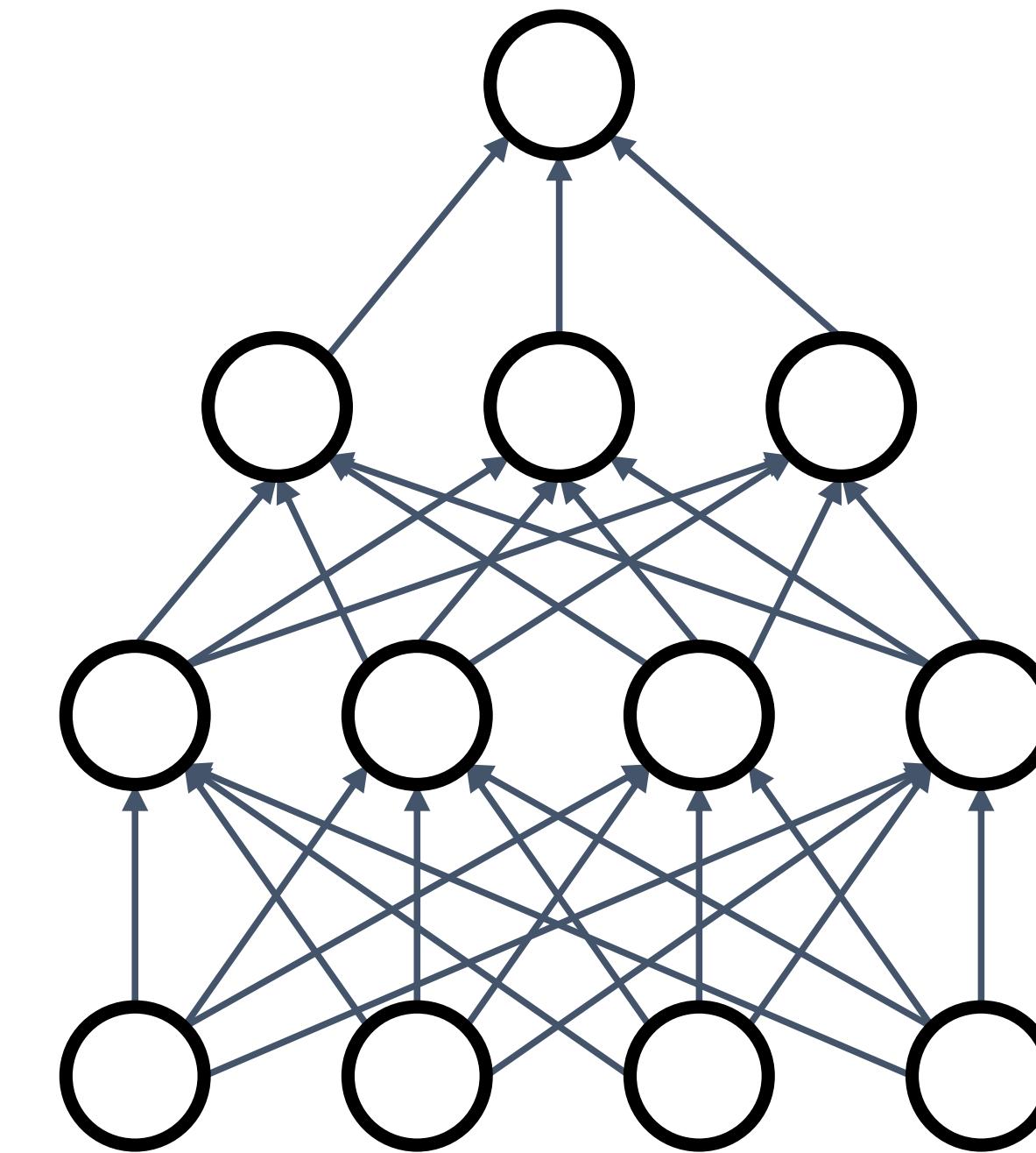
Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect



Regularization: Fractional Pooling

Training: Use randomized pooling regions

Testing: Average predictions over different samples

Examples:

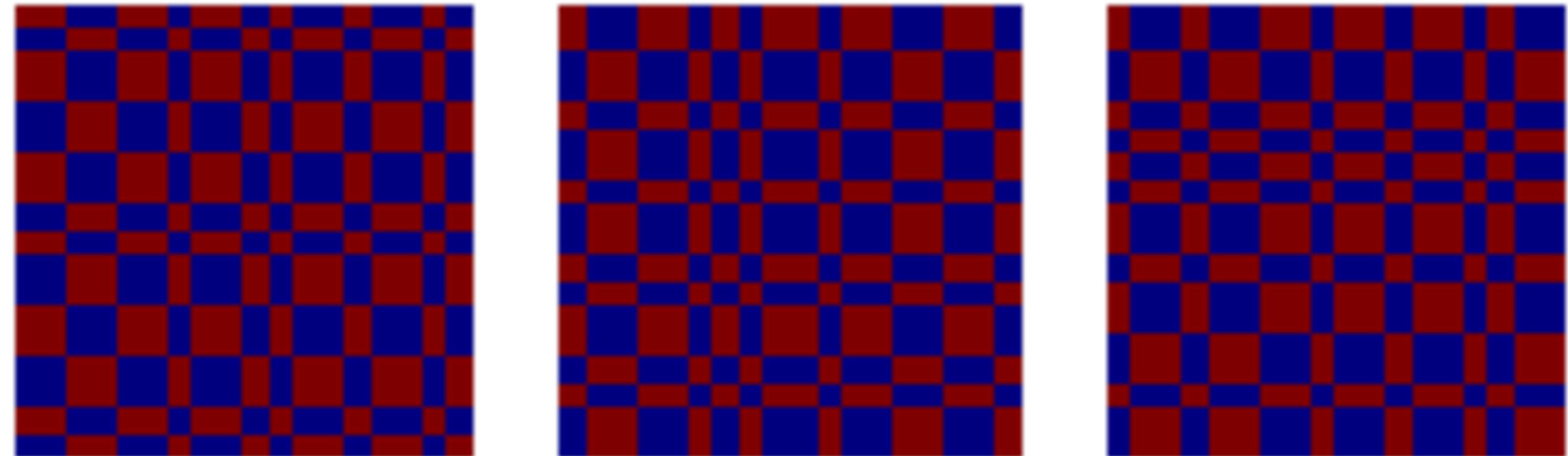
Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling



Regularization: Stochastic Depth

Training: Skip some residual blocks in ResNet

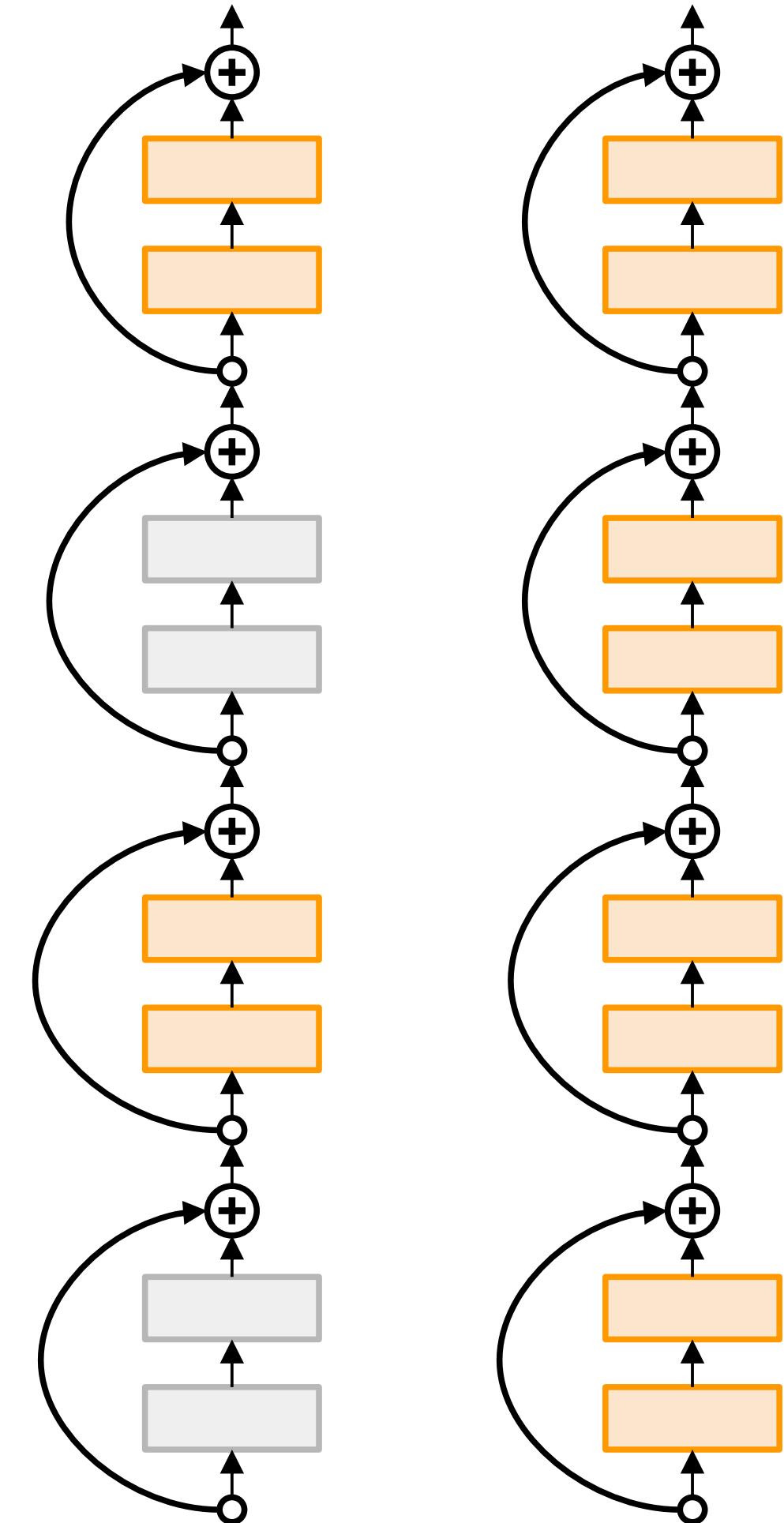
Testing: Use the whole network

Examples:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth

Starting to become common in recent architectures:

- Pham et al, “Very Deep Self-Attention Networks for End-to-End Speech Recognition”, INTERSPEECH 2019
- Tan and Le, “EfficientNetV2: Smaller Models and Faster Training”, ICML 2021
- Fan et al, “Multiscale Vision Transformers”, ICCV 2021
- Bello et al, “Revisiting ResNets: Improved Training and Scaling Strategies”, NeurIPS 2021
- Steiner et al, “How to train your ViT? Data, Augmentation, and Regularization in Vision Transformers”, arXiv 2021



Regularization: CutOut

Training: Set random image regions to 0

Testing: Use the whole image

Examples:

Dropout

Batch Normalization

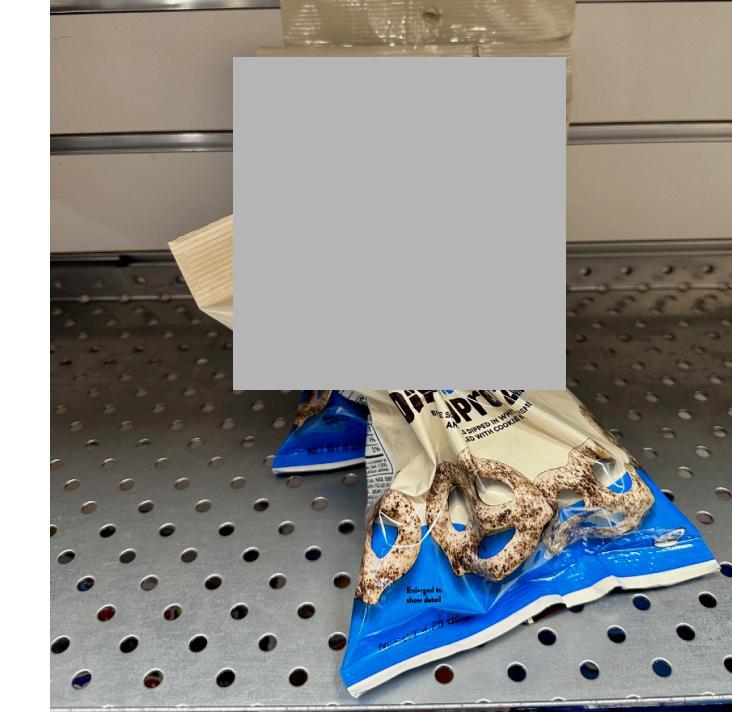
Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth

Cutout / Random Erasing



Replace random regions with
mean value or random values

Regularization: Mixup

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout

Batch Normalization

Data Augmentation

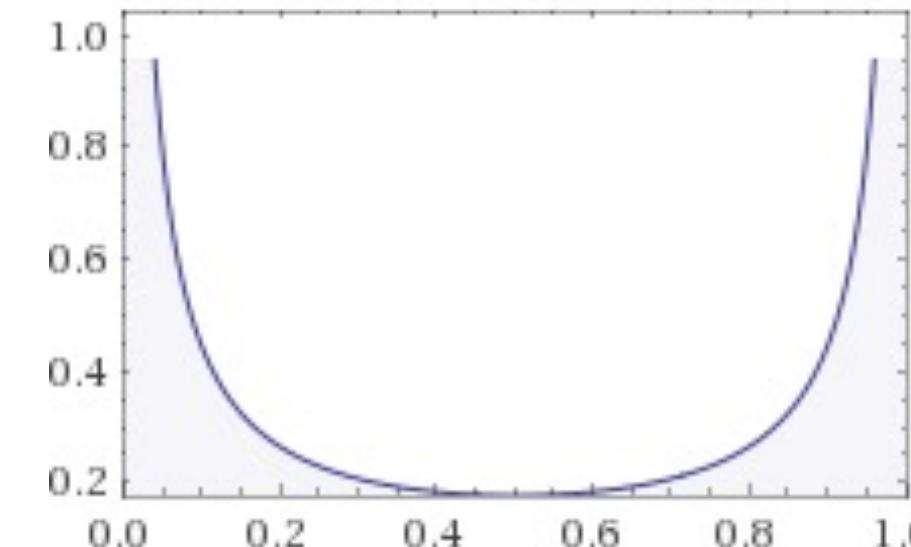
DropConnect

Fractional Max Pooling

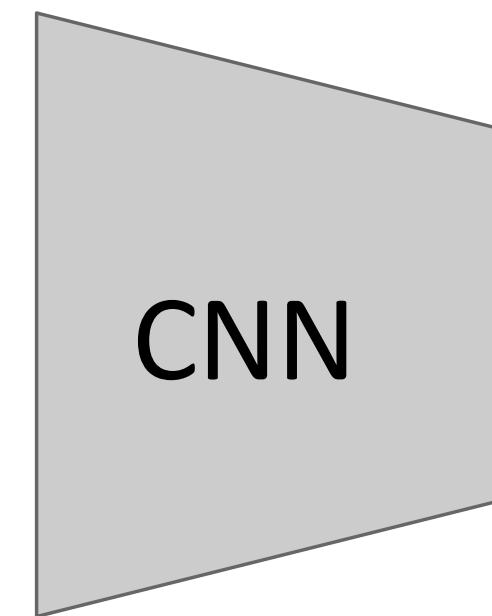
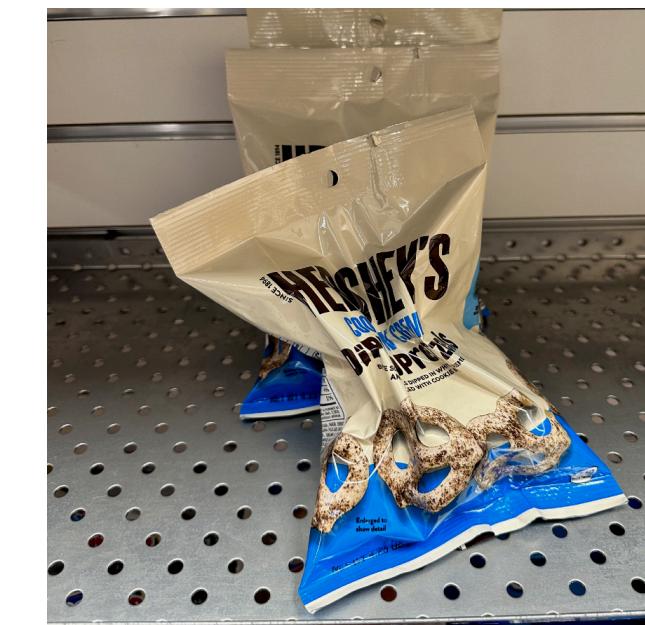
Stochastic Depth

Cutout / Random Erasing

Mixup



Sample blend probability from a beta distribution $\text{Beta}(a, b)$ with $a=b=0$ so blend weights are close to 0/1



Target label:
Pretzels: 0.6
Robot: 0.4



Randomly blend the pixels of pairs of training images, e.g.
60% pretzels, 40% robot

Regularization: CutMix

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout

Batch Normalization

Data Augmentation

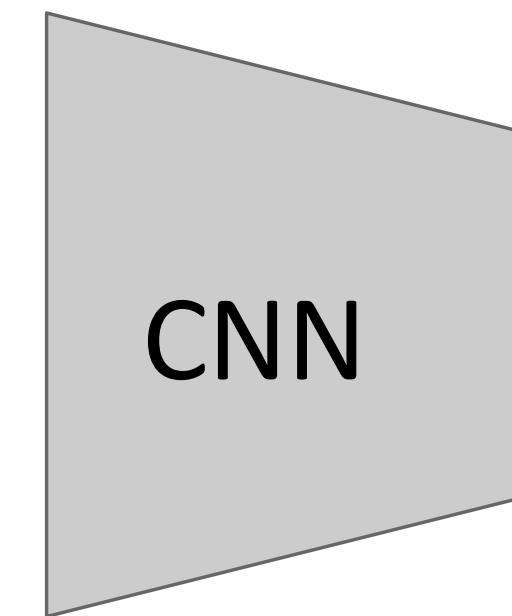
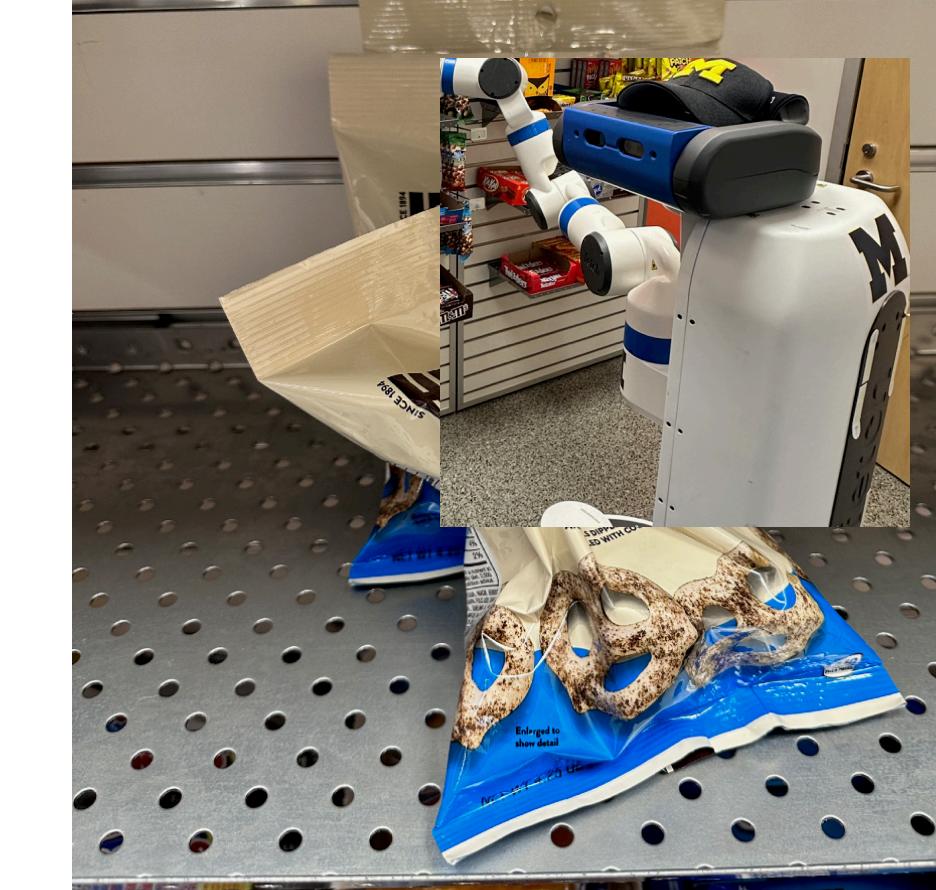
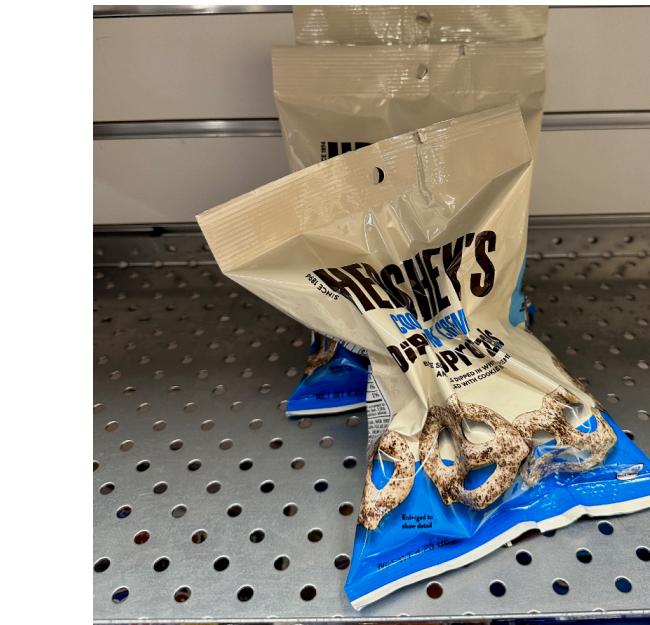
DropConnect

Fractional Max Pooling

Stochastic Depth

Cutout / Random Erasing

Mixup / CutMix



Target label:
Pretzels: 0.6
Robot: 0.4

Replace random crops of one image
with another, e.g. 60% of pixels
from pretzels, 40% from robot

Regularization: Label Smoothing

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect

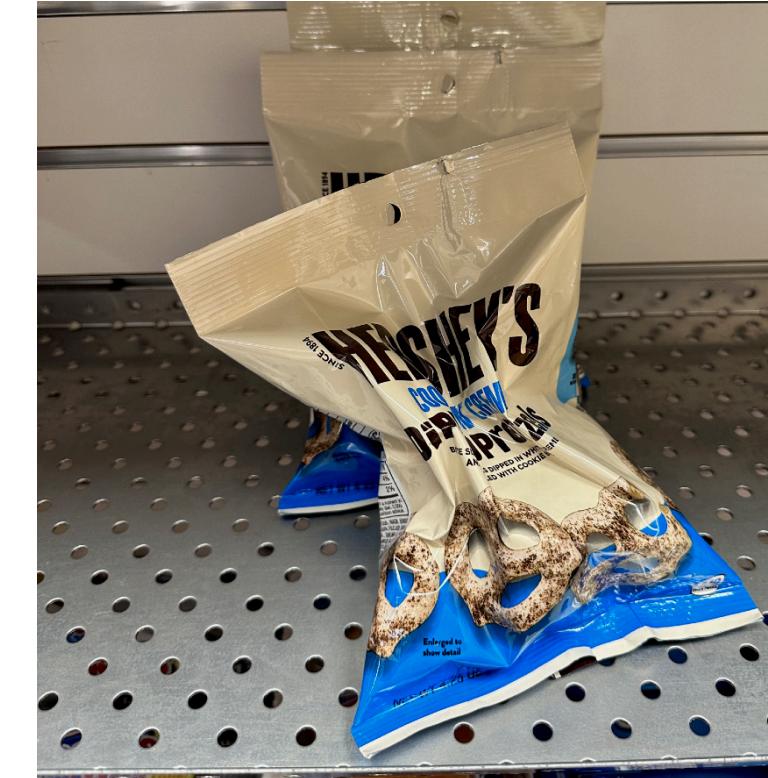
Fractional Max Pooling

Stochastic Depth

Cutout / Random Erasing

Mixup / CutMix

Label Smoothing



Standard Training

Pretzels: 100%

Robot: 0%

Sugar: 0%

Label Smoothing

Pretzels: 90%

Robot: 5%

Sugar: 5%

Set target distribution to be $1 - \frac{K-1}{K}\epsilon$ on the correct category and ϵ/K on all other categories, with K categories and $\epsilon \in (0,1)$.

Loss is cross-entropy between predicted and target distribution.

Regularization: Summary

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth

Cutout / Random Erasing

Mixup / CutMix

Label Smoothing

- Use DropOut for large fully-connected layers
- Data augmentation is always a good idea
- Use BatchNorm for CNNs (but not ViTs)
- Try Cutout, Mixup, CutMix, Stochastic Depth, Label Smoothing to squeeze out a bit of extra performance



Recap

1. One time setup:

- Activation functions, data preprocessing, weight initialization, regularization

2. Training dynamics:

Today

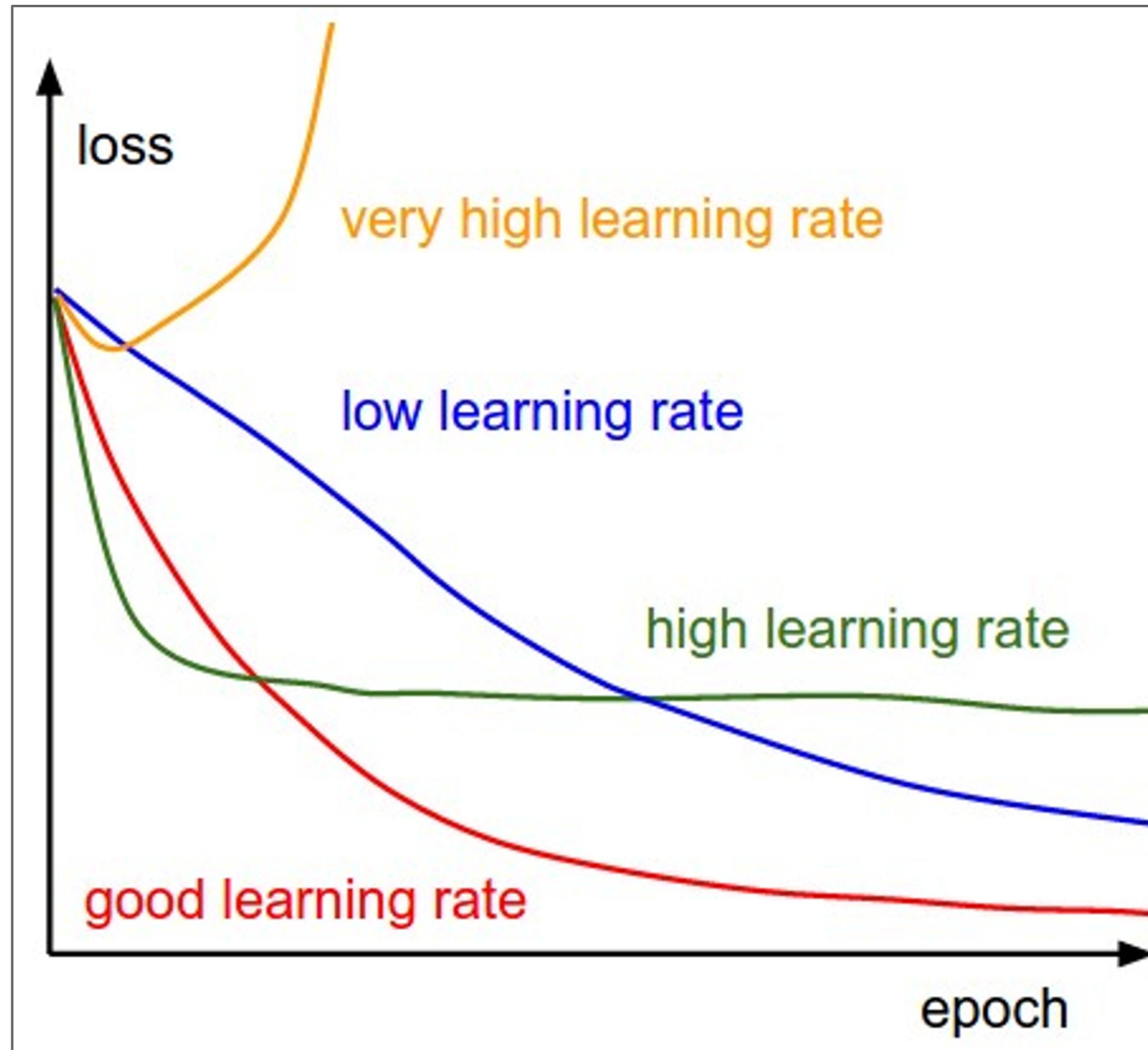
- Learning rate schedules; large-batch training; hyperparameter optimization

3. After training:

- Model ensembles, transfer learning

Learning Rate Schedules

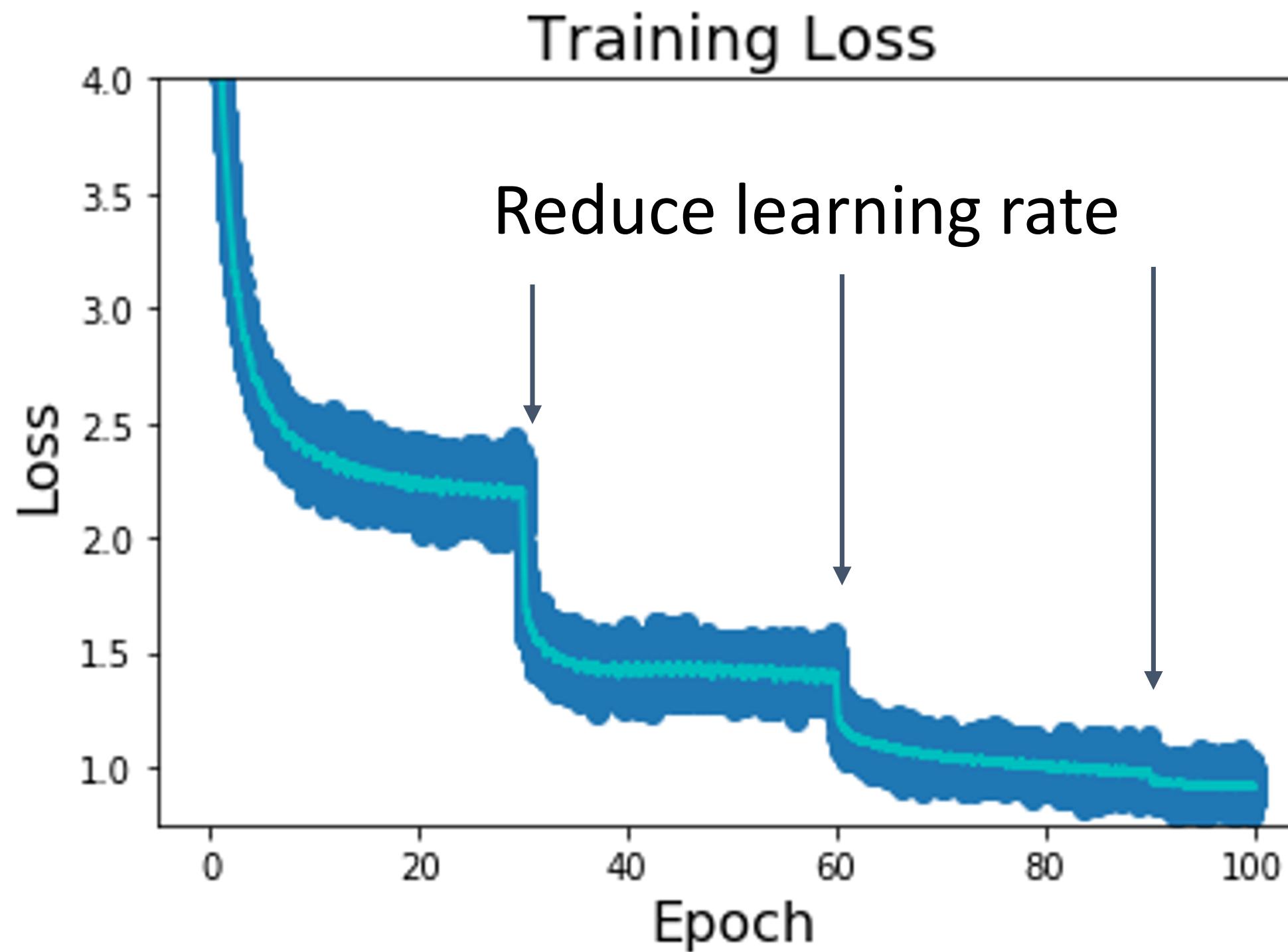
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as hyper parameter



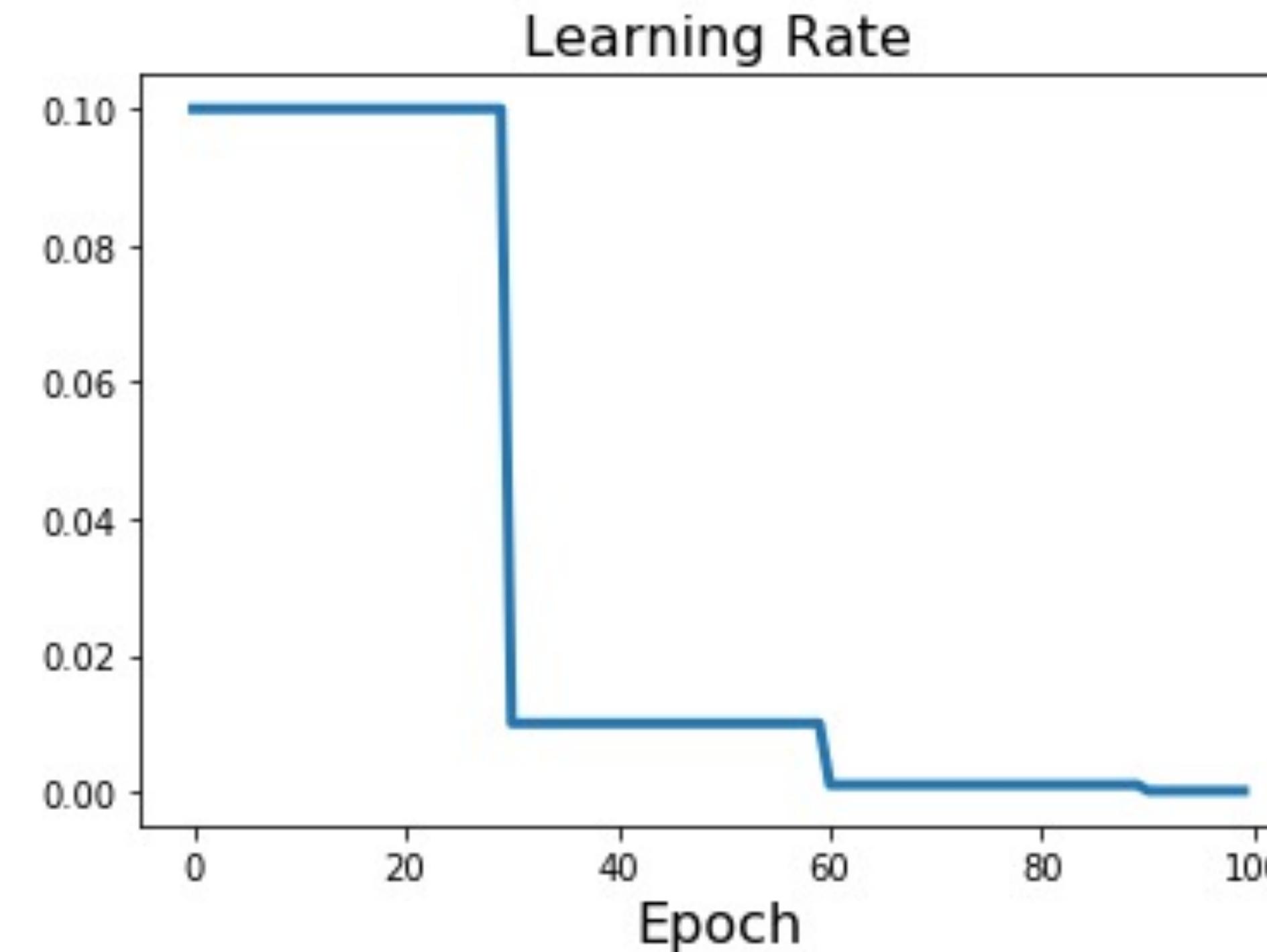
Q: Which one of these learning rates is best to use?

A: All of them! Start with large learning rate and decay over time.

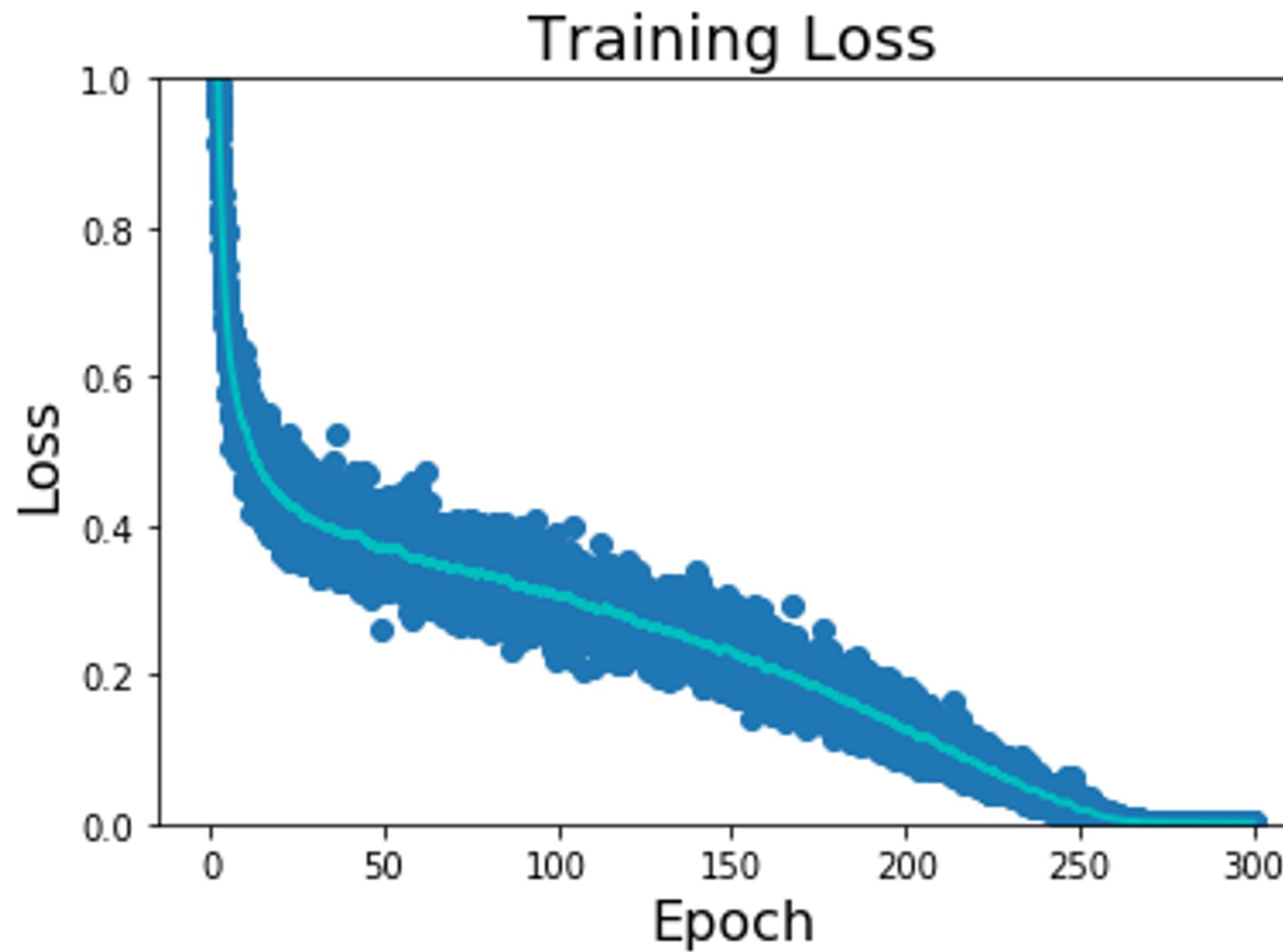
Learning Rate Decay: Step



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

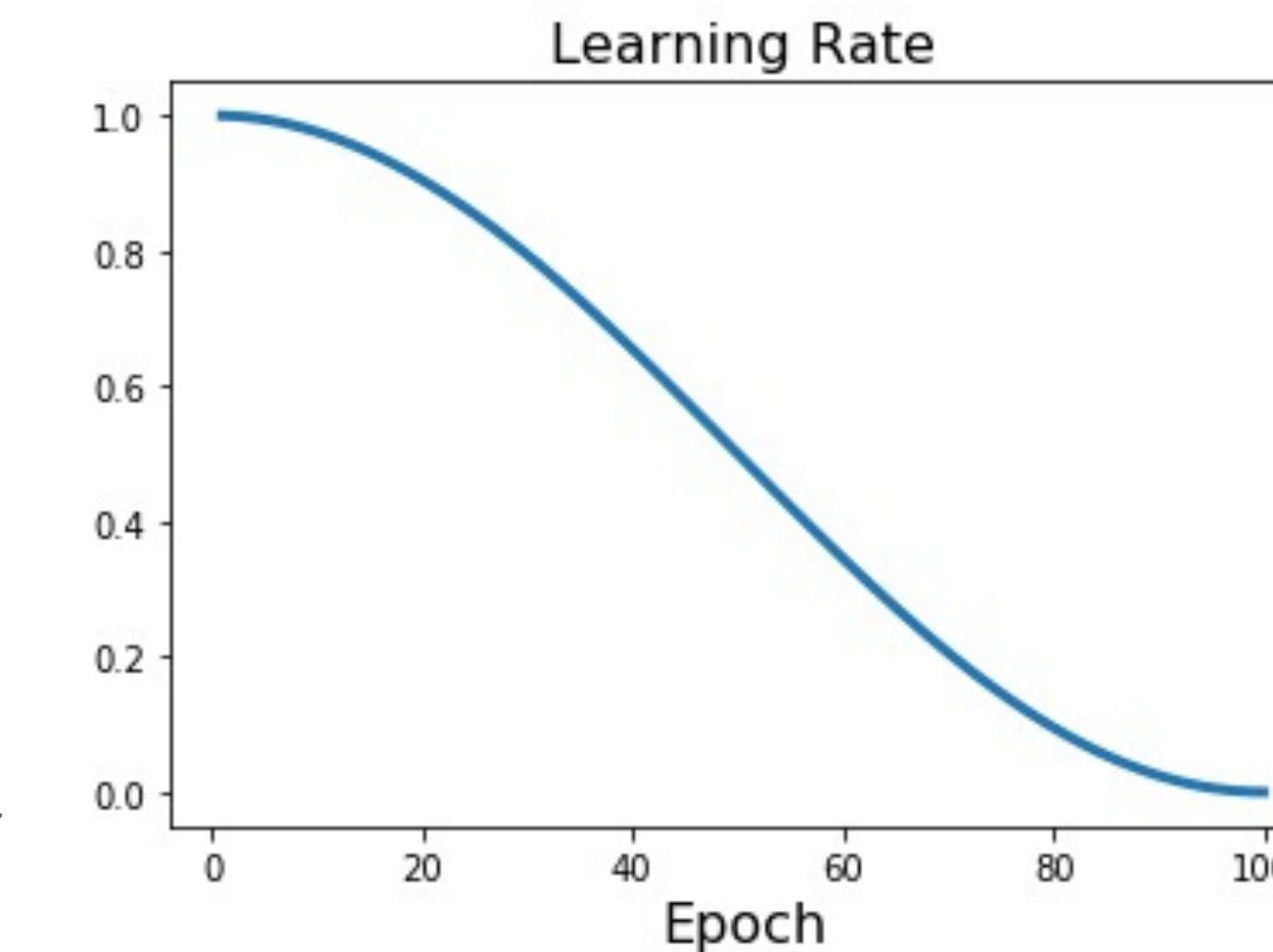


Learning Rate Decay: Cosine

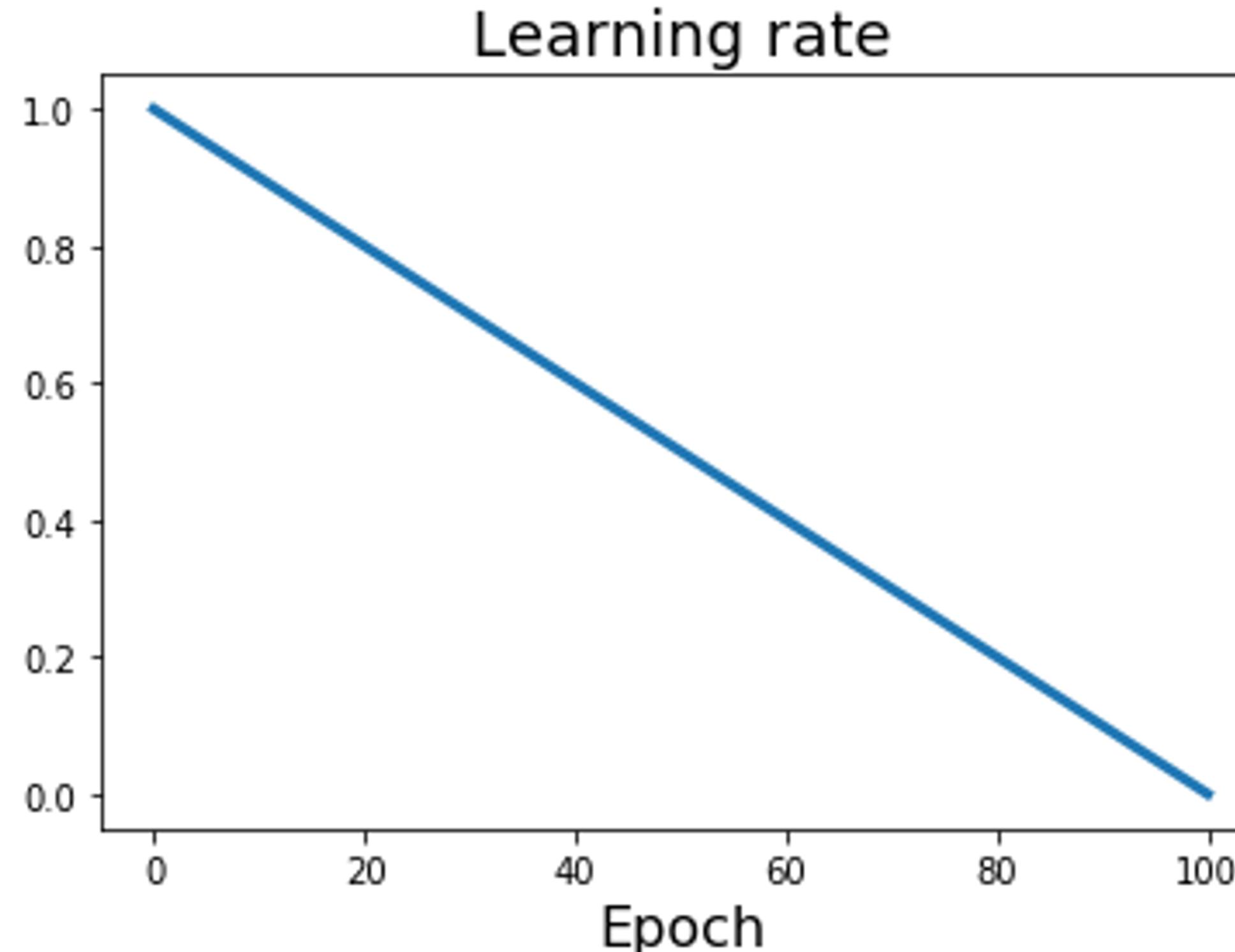


Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0(1 + \cos(\frac{t\pi}{T}))$



Learning Rate Decay: Linear



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0(1 + \cos(\frac{t\pi}{T}))$

Linear: $\alpha_t = \alpha_0(1 - \frac{t}{T})$

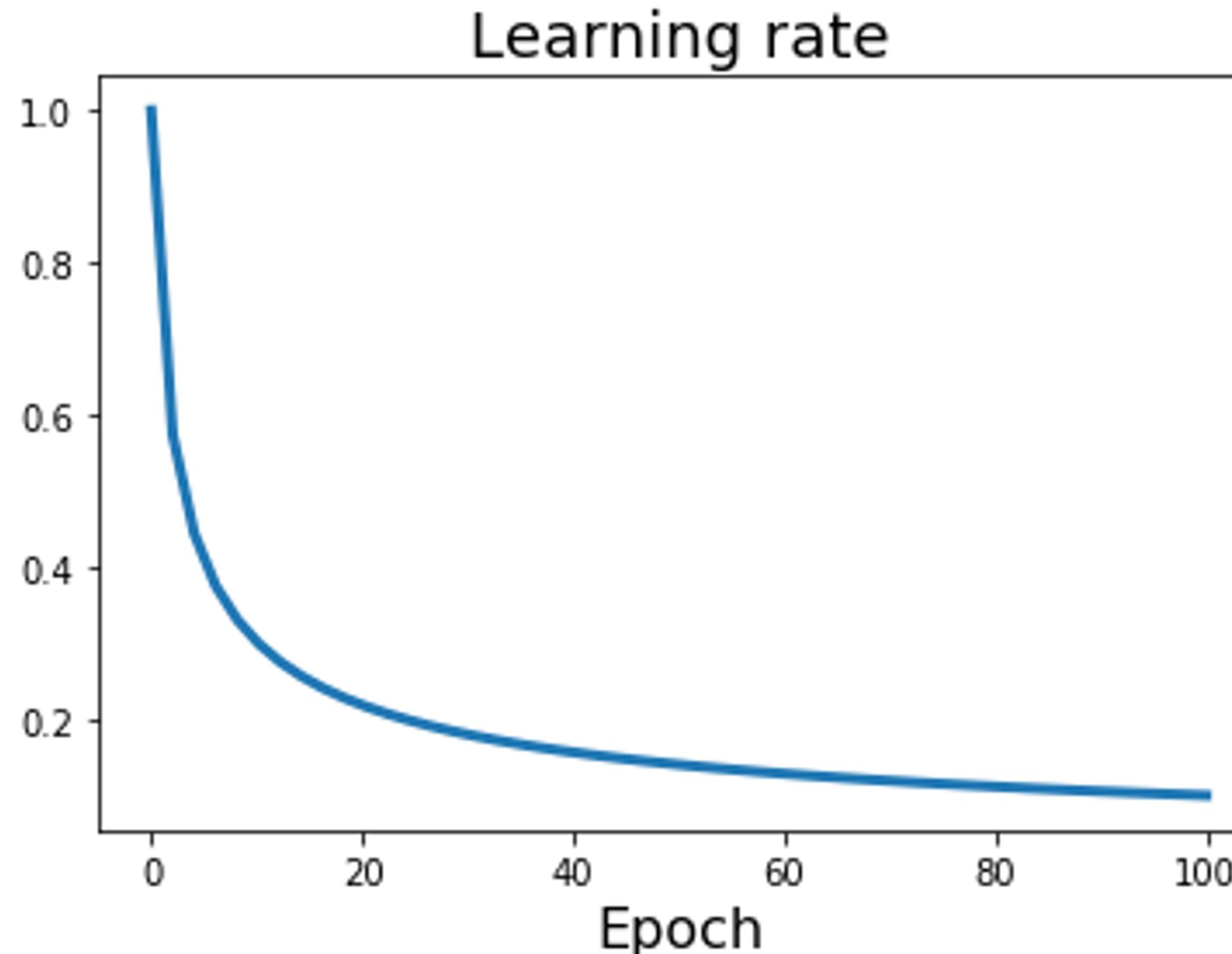


Devlin et al, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", NAACL 2018

Liu et al, "RoBERTa: A Robustly Optimized BERT Pretraining Approach", 2019

Yang et al, "XLNet: Generalized Autoregressive Pretraining for Language Understanding", NeurIPS 2019

Learning Rate Decay: Inverse Sqrt



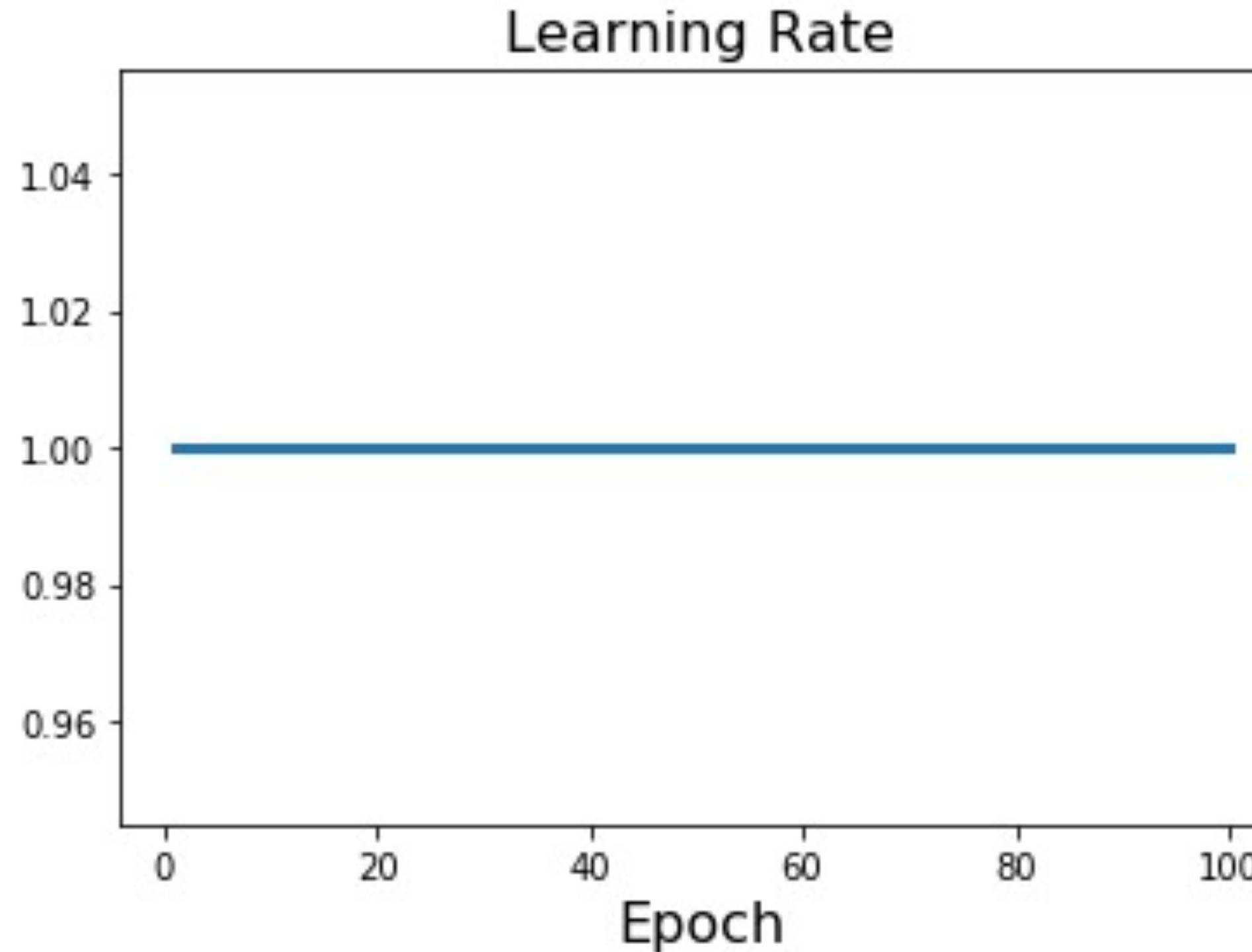
Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0(1 + \cos(\frac{t\pi}{T}))$

Linear: $\alpha_t = \alpha_0(1 - \frac{t}{T})$

Inverse sqrt: $\alpha_t = \alpha_0/\sqrt{t}$

Learning Rate Decay: Constant!



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0(1 + \cos(\frac{t\pi}{T}))$

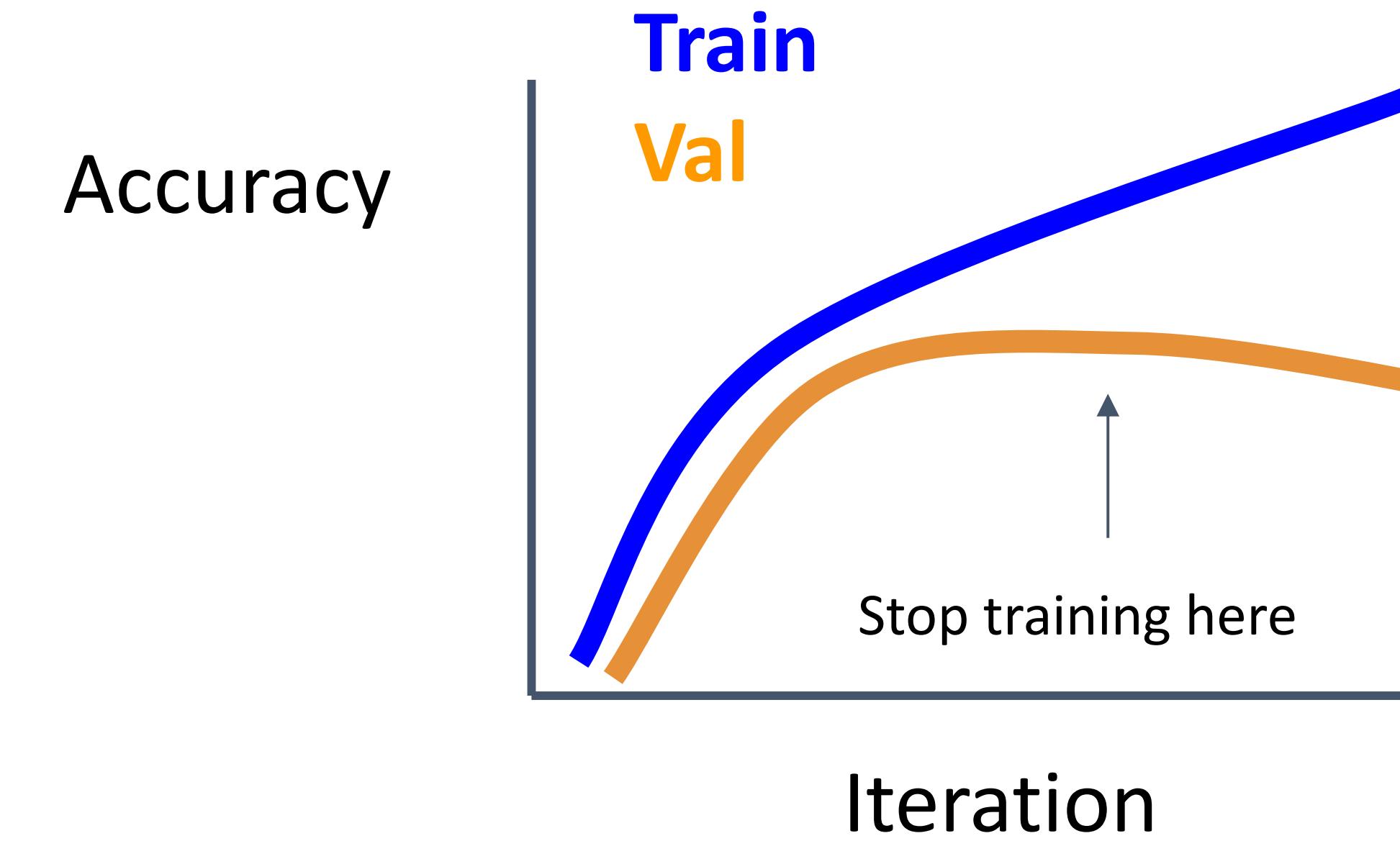
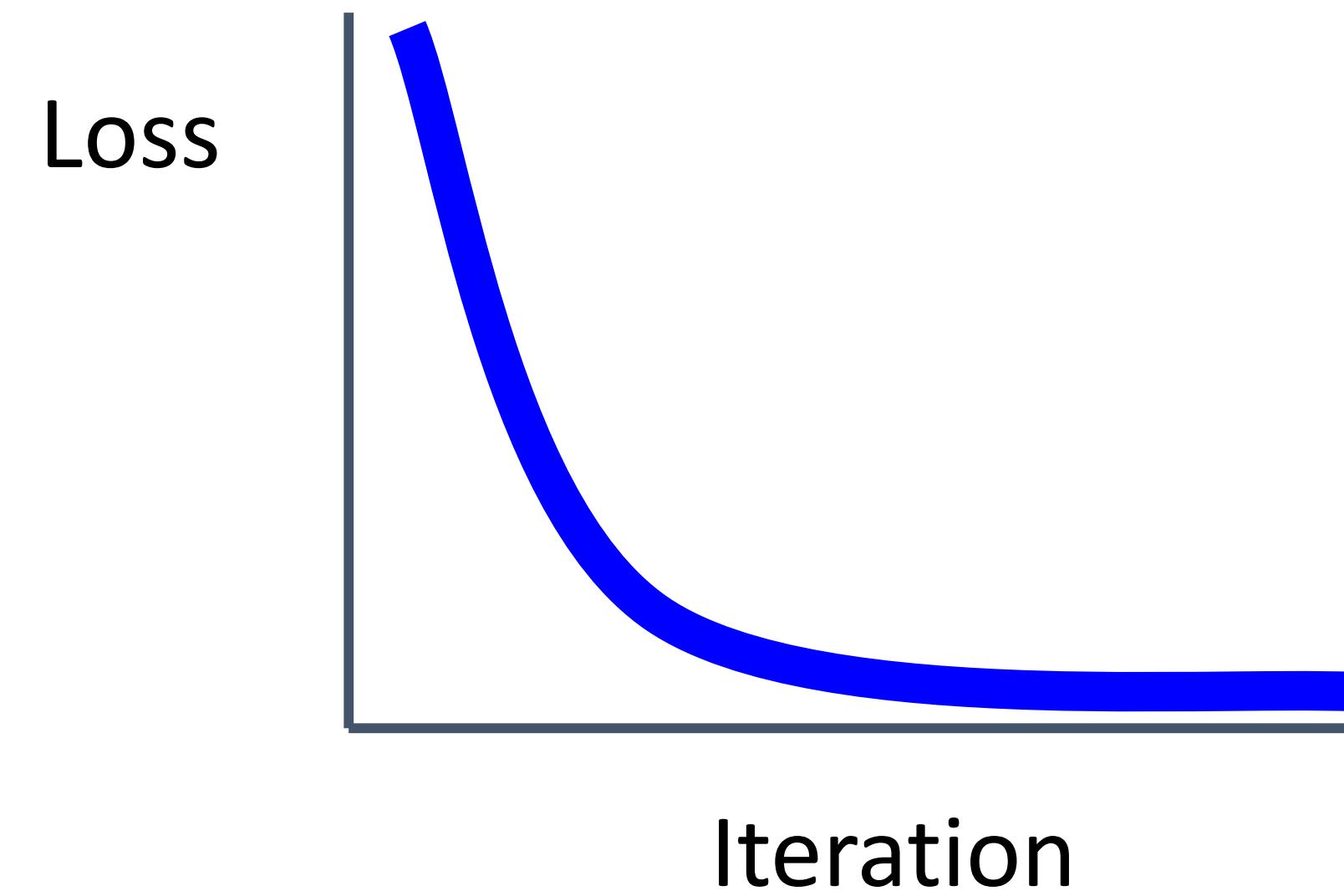
Linear: $\alpha_t = \alpha_0(1 - \frac{t}{T})$

Inverse sqrt: $\alpha_t = \alpha_0/\sqrt{t}$

Constant: $\alpha_t = \alpha_0$



How long to train? Early Stopping



Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot that
worked best on val. **Always a good idea to do this!**

Choosing Hyperparameters



Choosing Hyperparameters: Grid Search

Choose several values for each hyper parameter
(Often space choices log-linearly)

Example:

Weight decay: $[1 \times 10^{-4}, 1 \times 10^{-3}, 1 \times 10^{-2}, 1 \times 10^{-1}]$

Learning rate: $[1 \times 10^{-4}, 1 \times 10^{-3}, 1 \times 10^{-2}, 1 \times 10^{-1}]$

Evaluate all possible choices on this **hyperparameter grid**

Choosing Hyperparameters: Random Search

Choose several values for each hyper parameter
(Often space choices log-linearly)

Example:

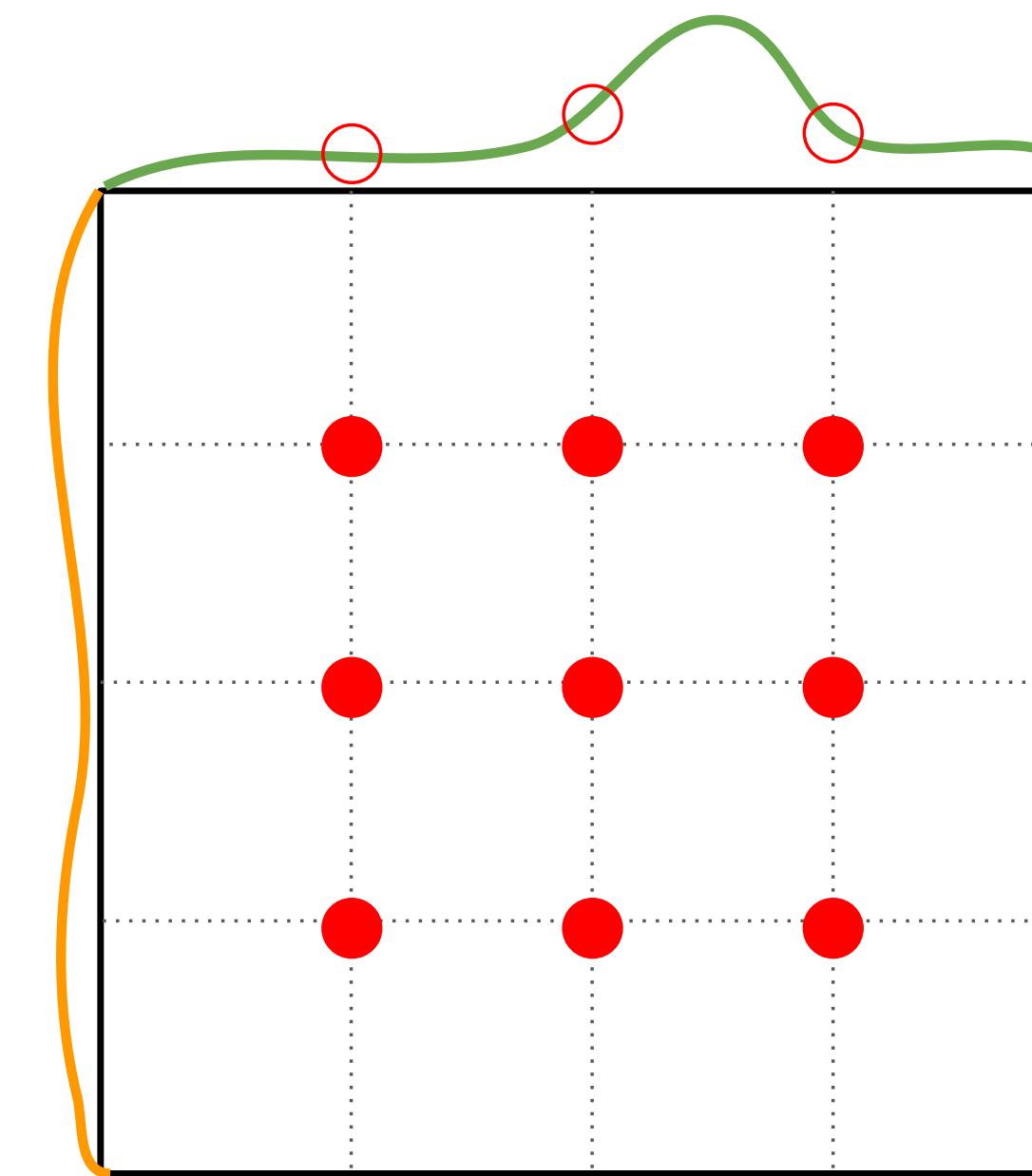
Weight decay: $[1 \times 10^{-4}, 1 \times 10^{-1}]$

Learning rate: $[1 \times 10^{-4}, 1 \times 10^{-1}]$

Run many different trials

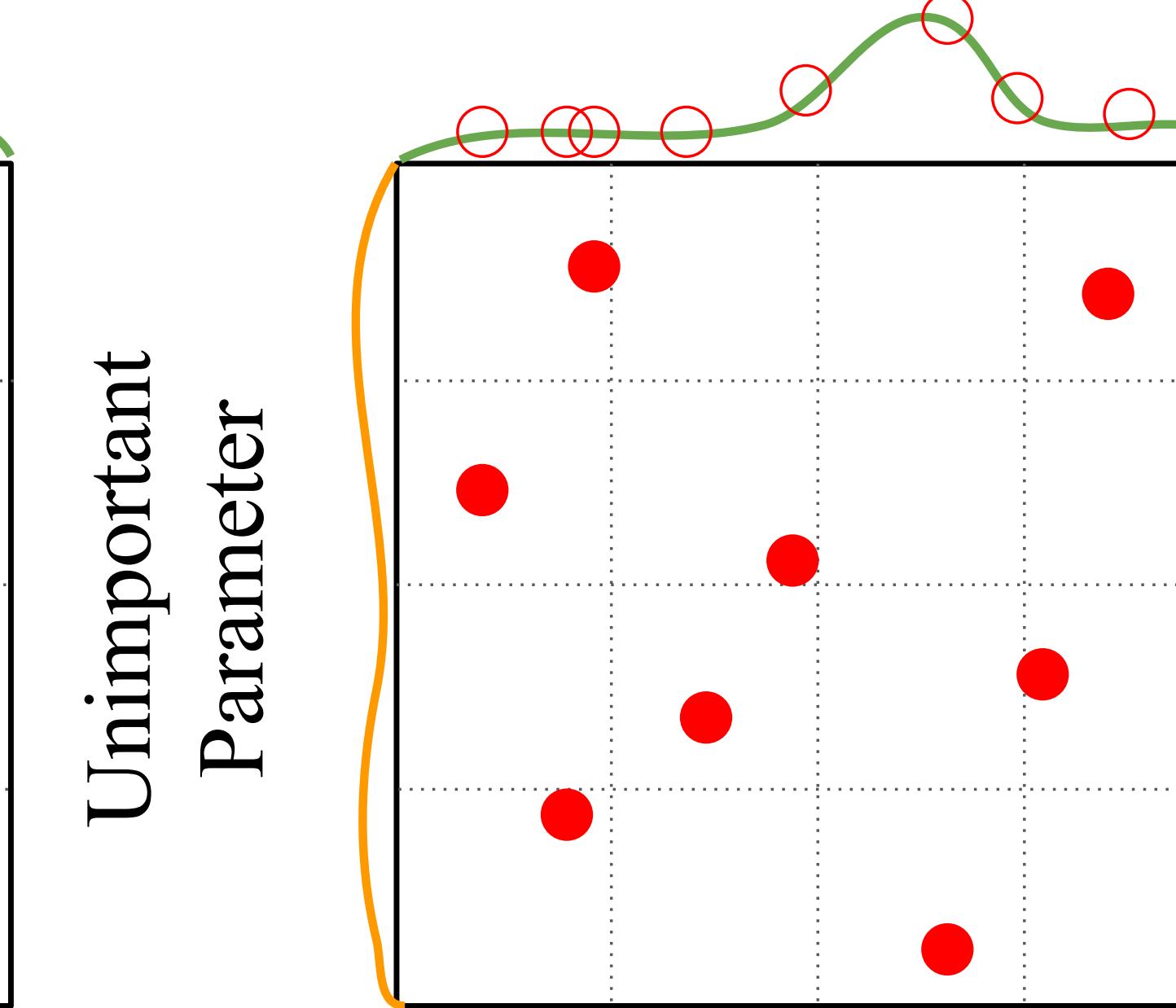
Hyperparameters: Random vs Grid Search

Grid Layout



Important
Parameter

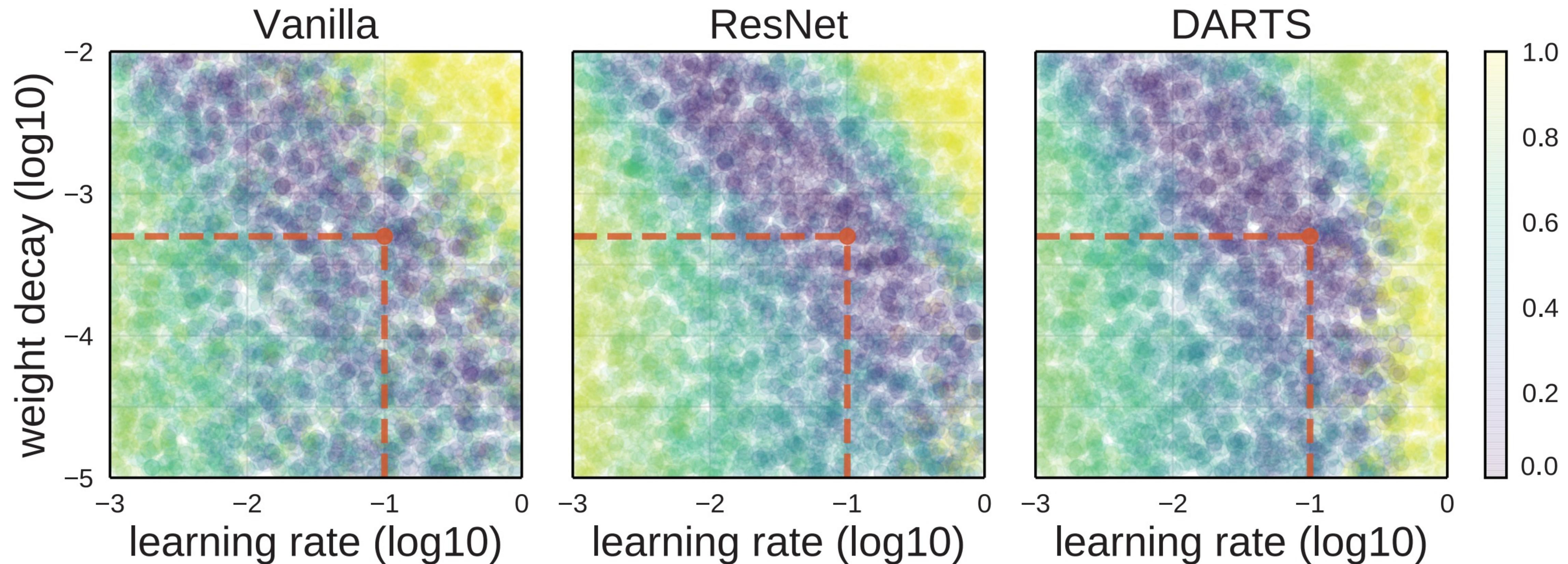
Random Layout



Important
Parameter

Unimportant
Parameter

Choosing Hyperparameters: Random Search



Choosing Hyperparameters (without tons of GPUs)



Choosing Hyperparameters

Step 1: Check initial loss

Turn off weight decay, sanity check loss at initialization
e.g. $\log(C)$ for softmax with C classes



Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Try to train to 100% training accuracy on a small sample of training data (~5-10 mini batches); fiddle with architecture, learning rate, weight initialization. Turn off regularization.

Loss not going down? LR too low, bad initialization

Loss explodes to Inf or NaN? LR too high, bad initialization



Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations

Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4



Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs

Good learning rates to try: 1e-4, 1e-5, 0



Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer

Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay



Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

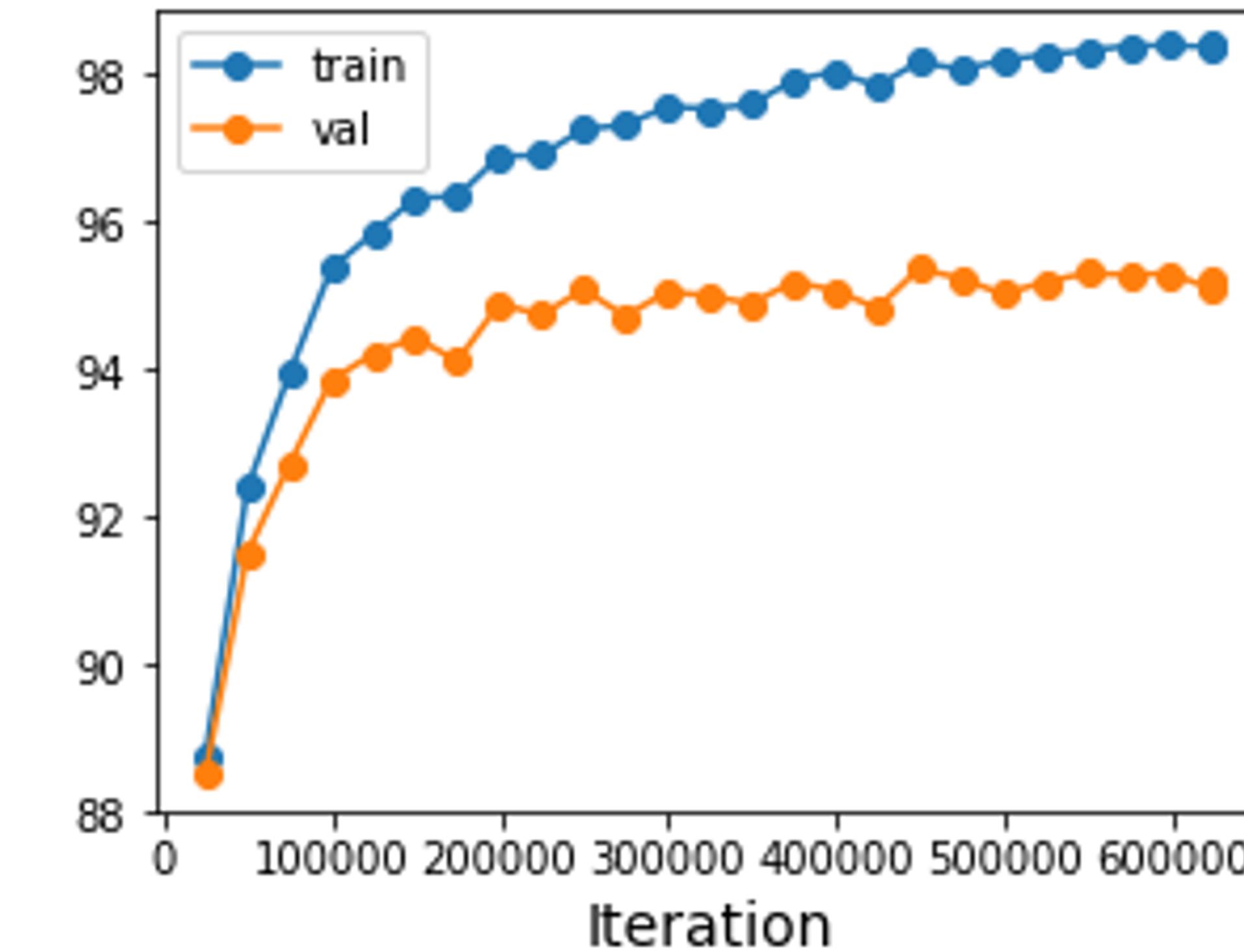
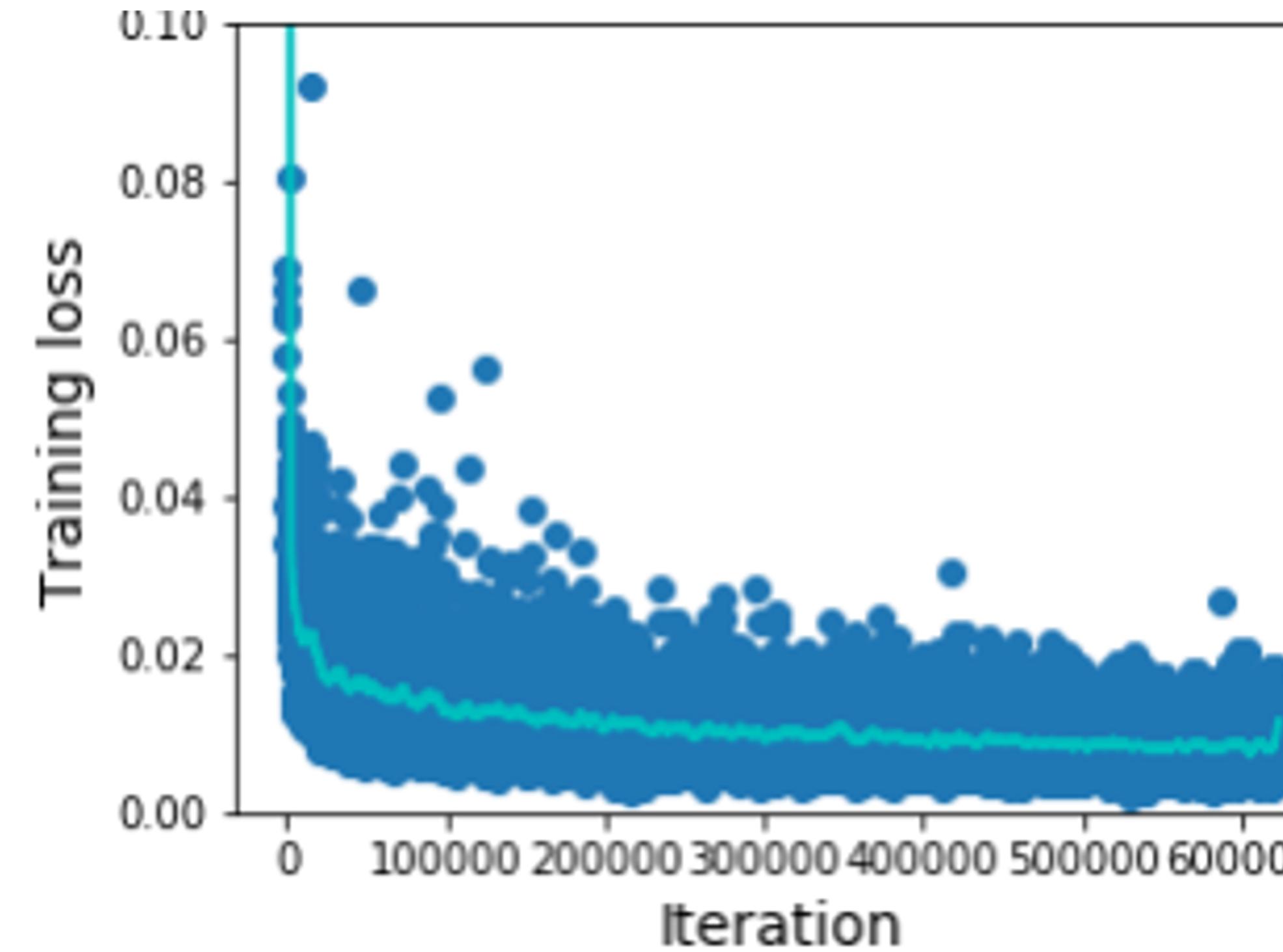
Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer

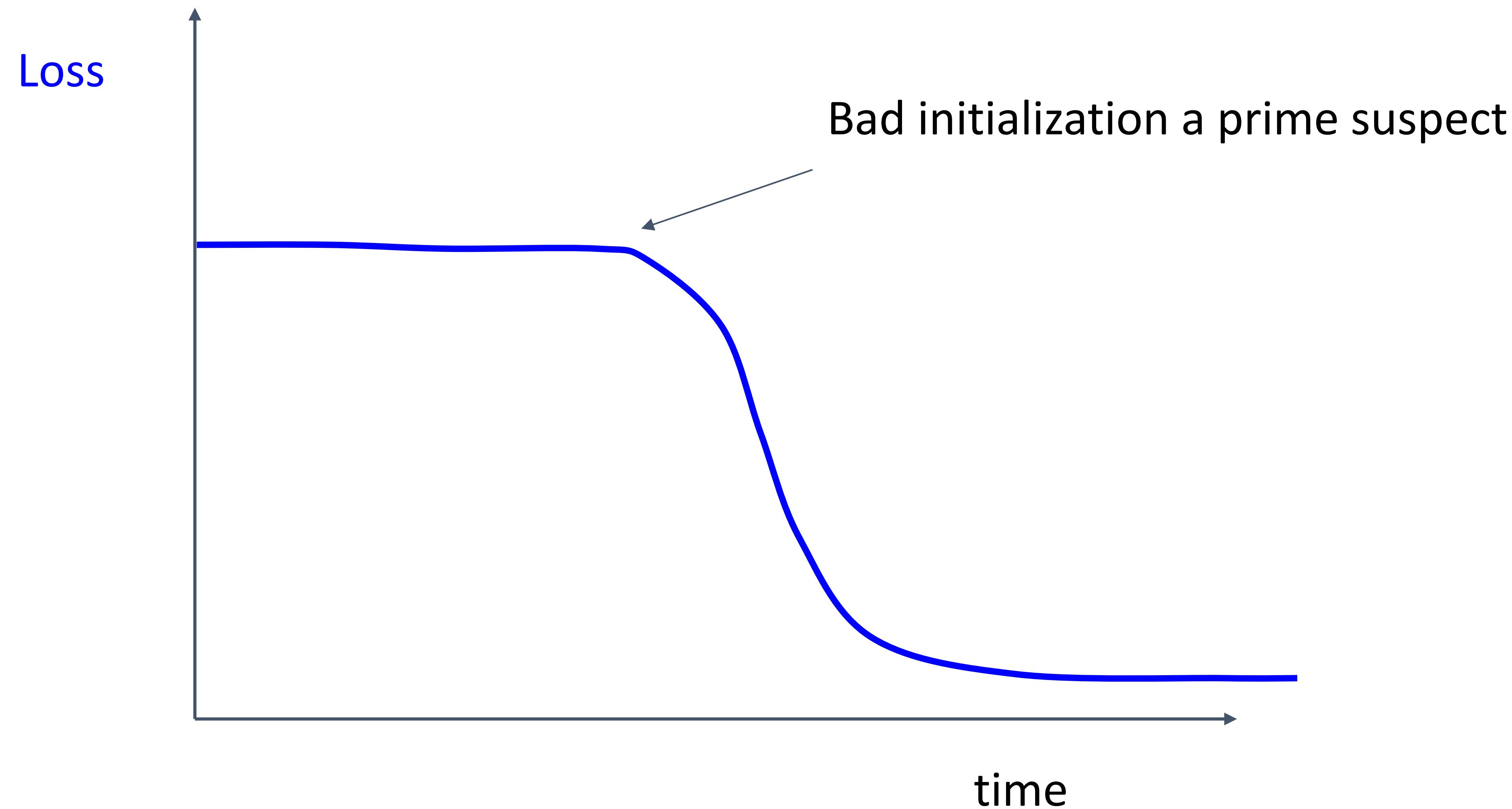
Step 6: Look at learning curves

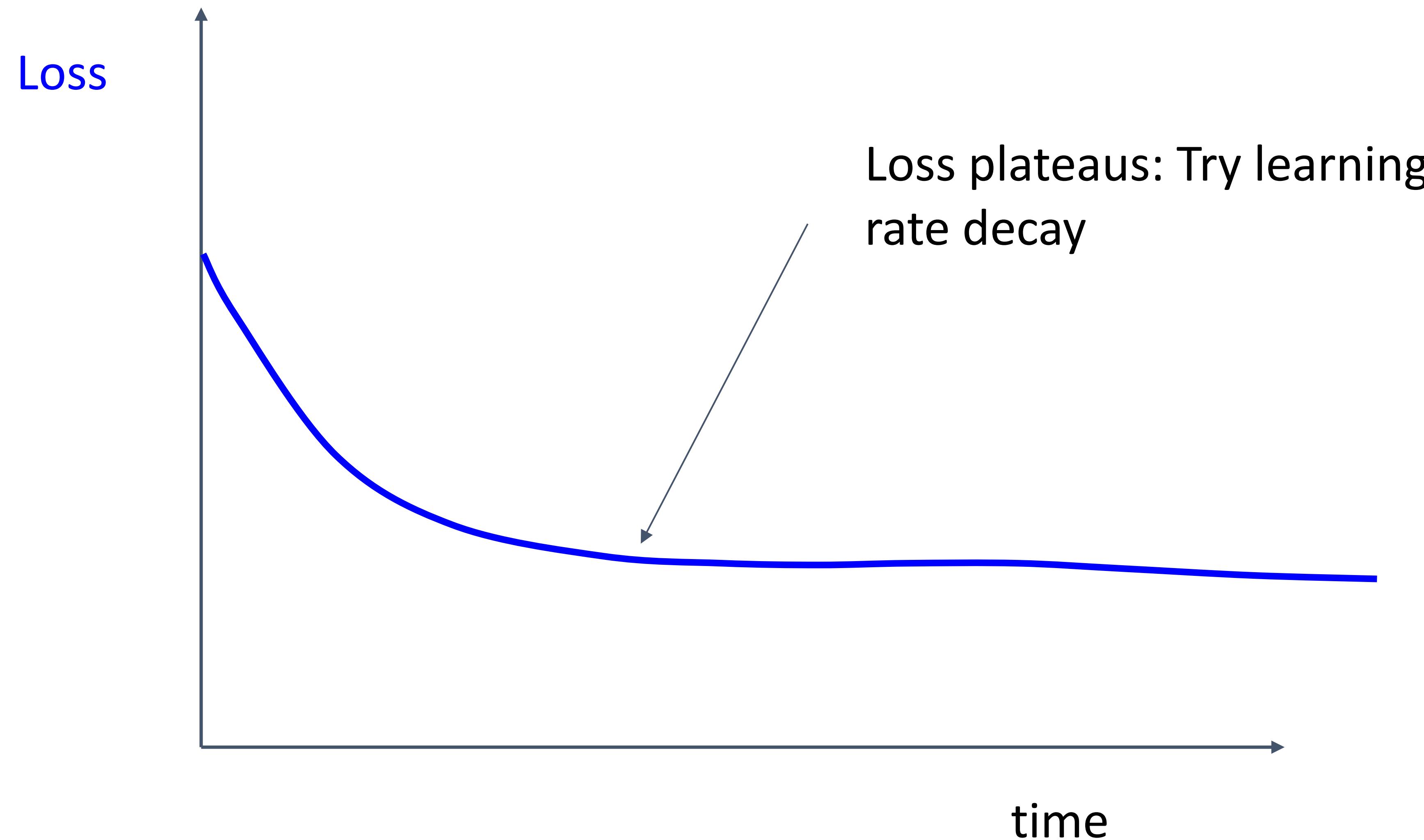


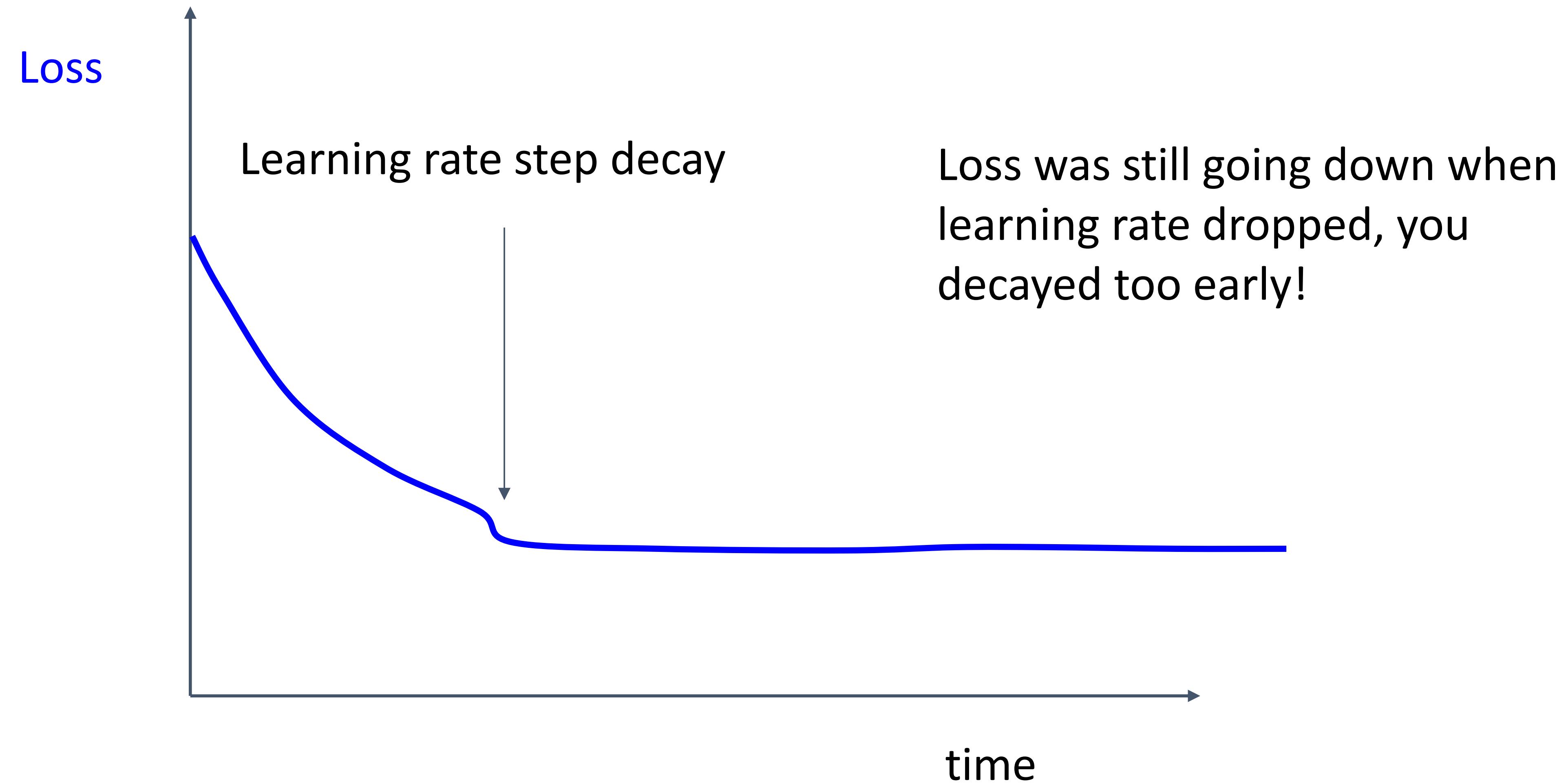
Look at Learning Curves!

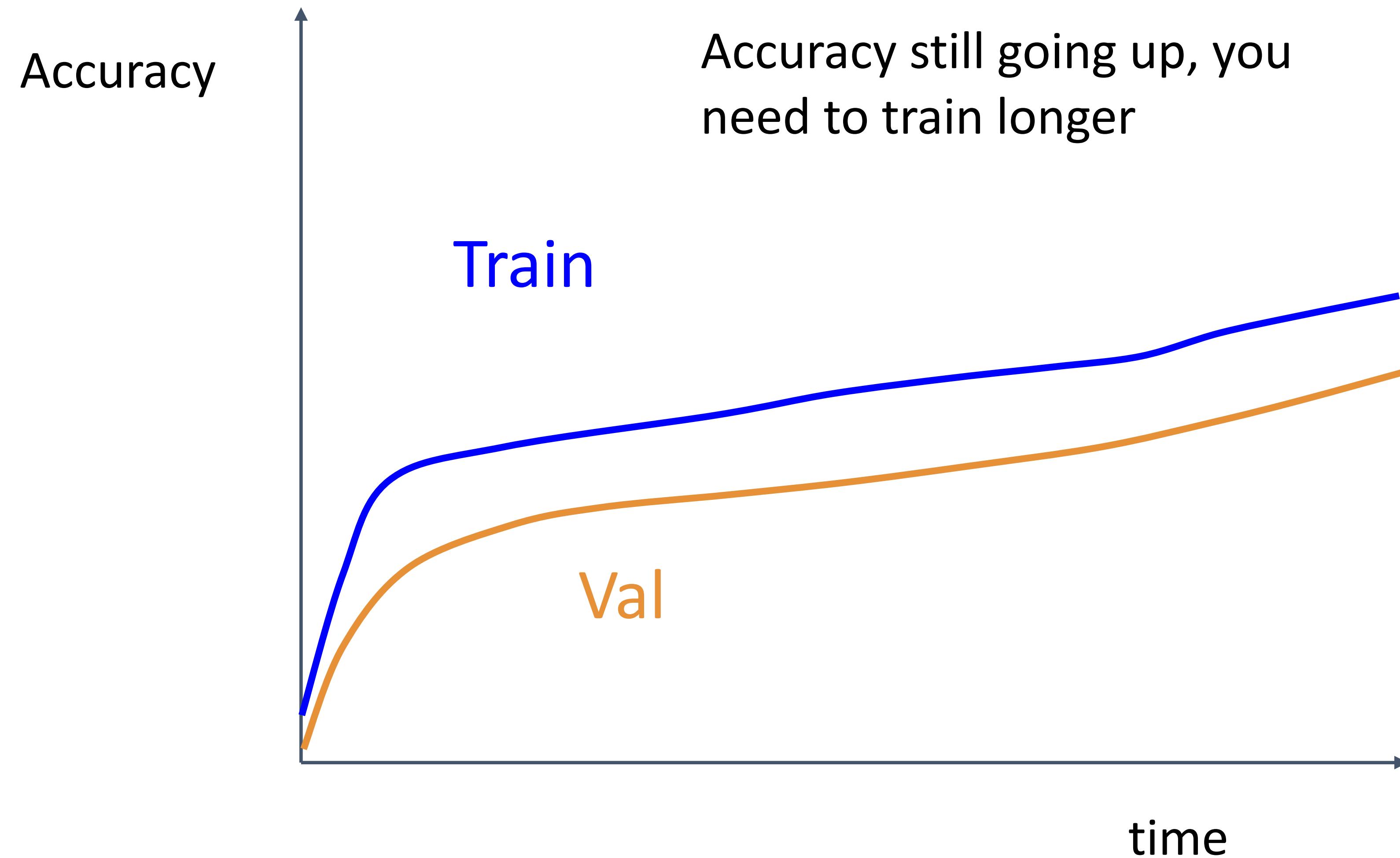


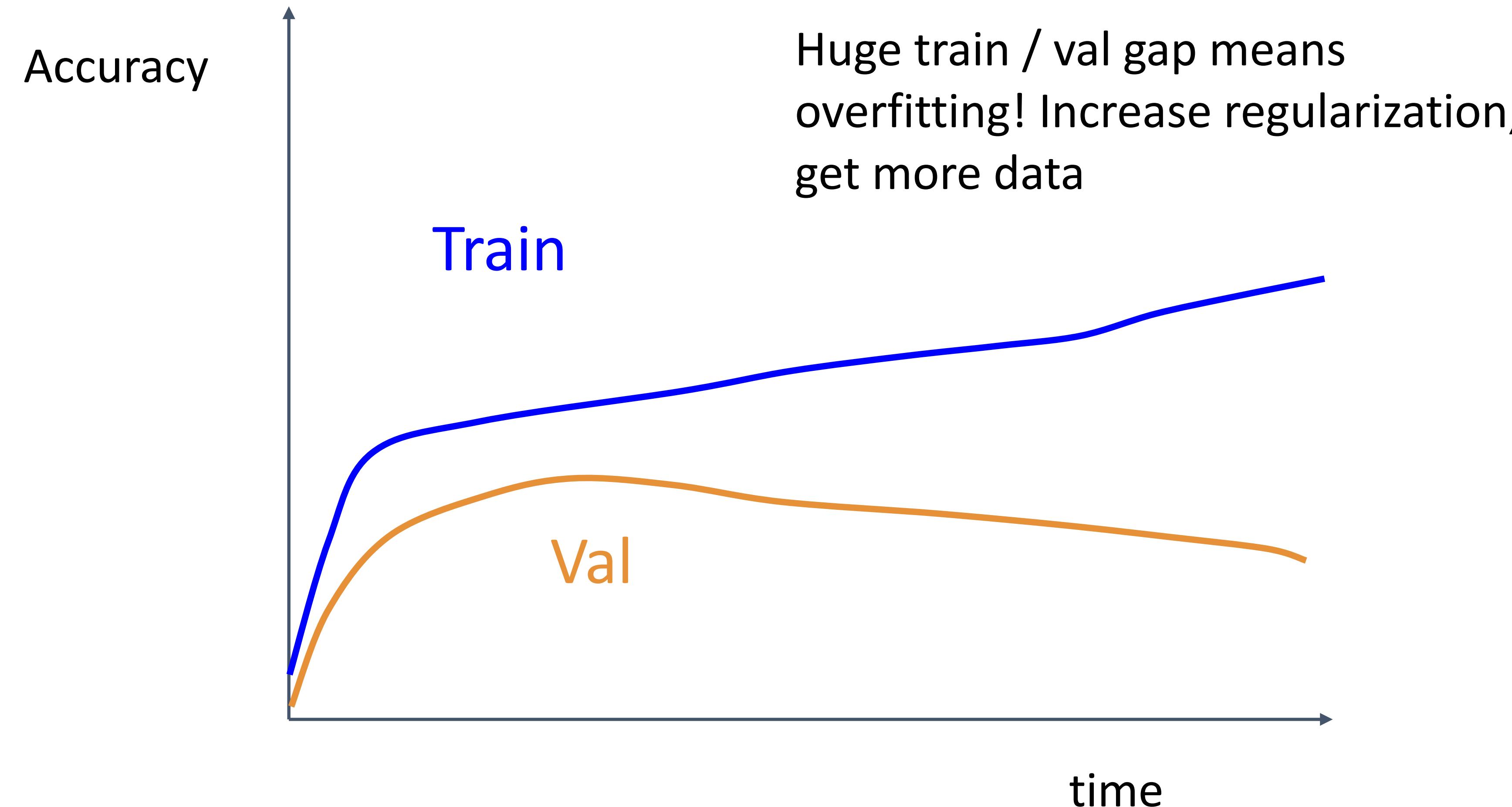
Losses may be noisy, use a scatter plot and also plot moving average to see trends better

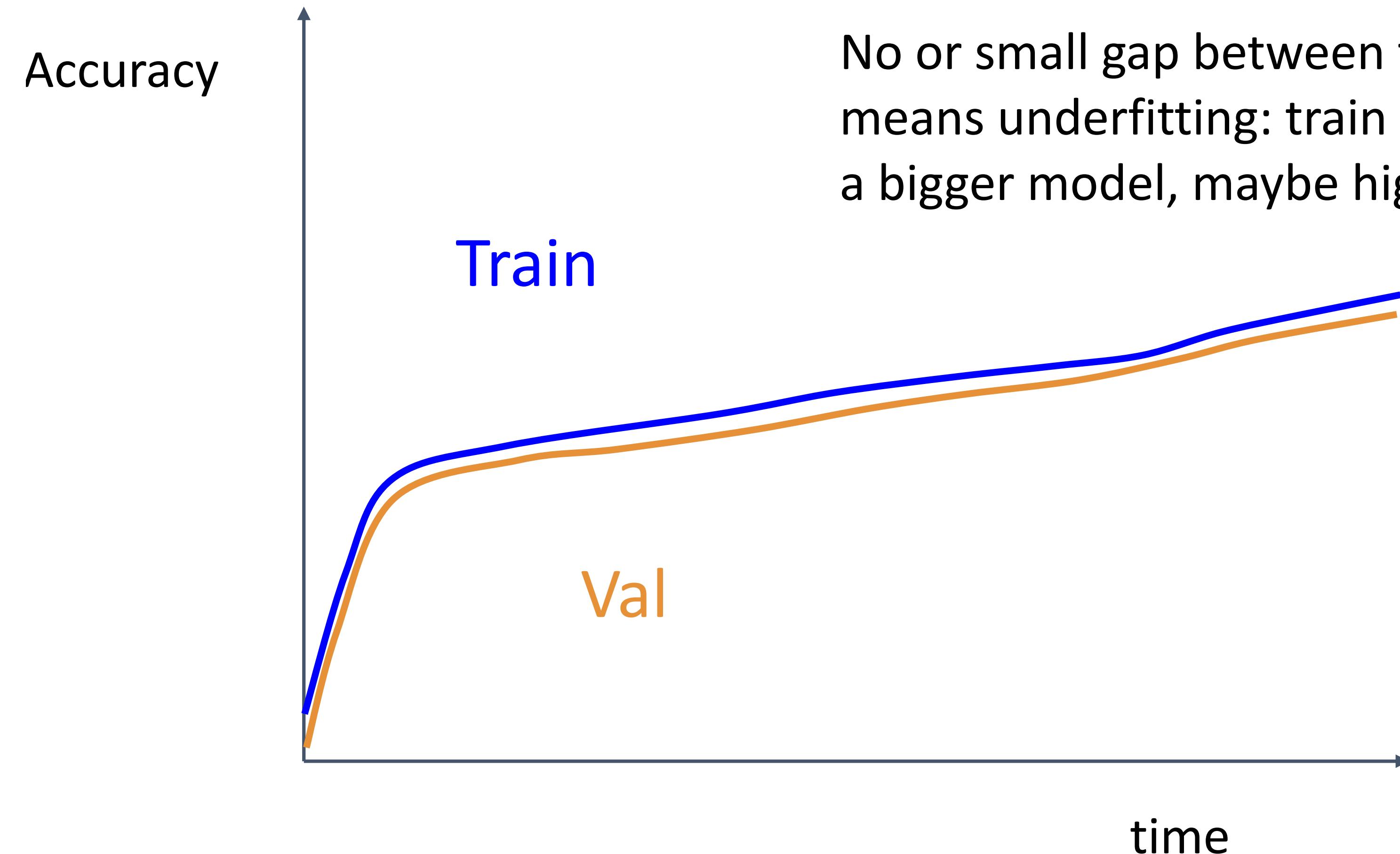












Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer

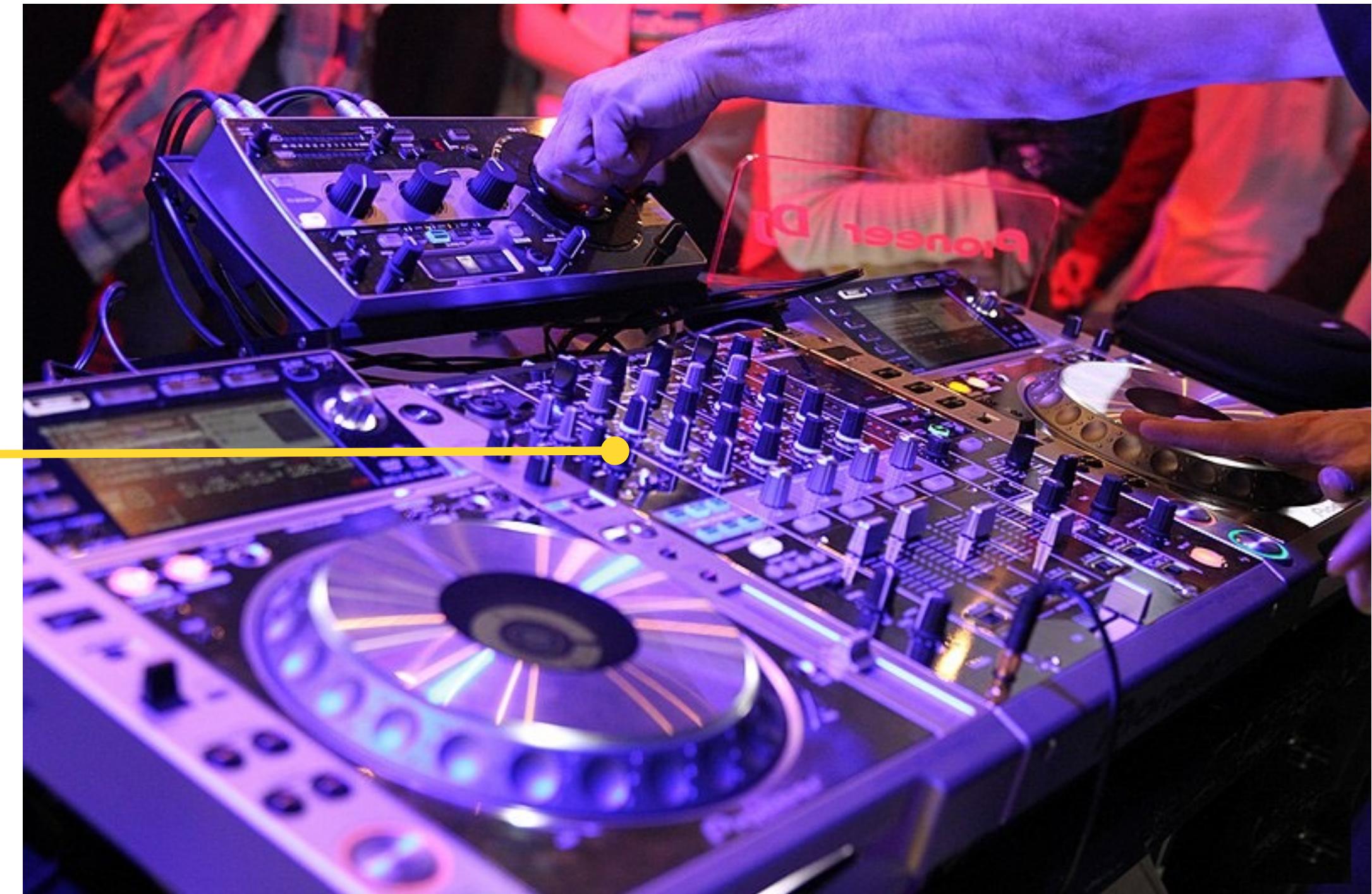
Step 6: Look at ~~learning curves~~ loss curves

Step 7: GOTO step 5

Hyperparameters to play with:

- Network architecture
- Learning rate, its decay schedule, update type
- Regularization (L2/ Dropout strength)

Neural networks practitioner
Music = loss function



Cross-validation “command center”



Track ratio of weight update / weight magnitude

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

Ratio between the updates and values: $\sim 0.0002 / 0.02 = 0.01$ (about okay)
want this to be somewhere around 0.001 or so

Overview

1. One time setup:

- Activation functions, data preprocessing, weight initialization, regularization

2. Training dynamics:

- Learning rate schedules; hyperparameter optimization

3. After training:

- Model ensembles, transfer learning, large-batch training

Model Ensembles

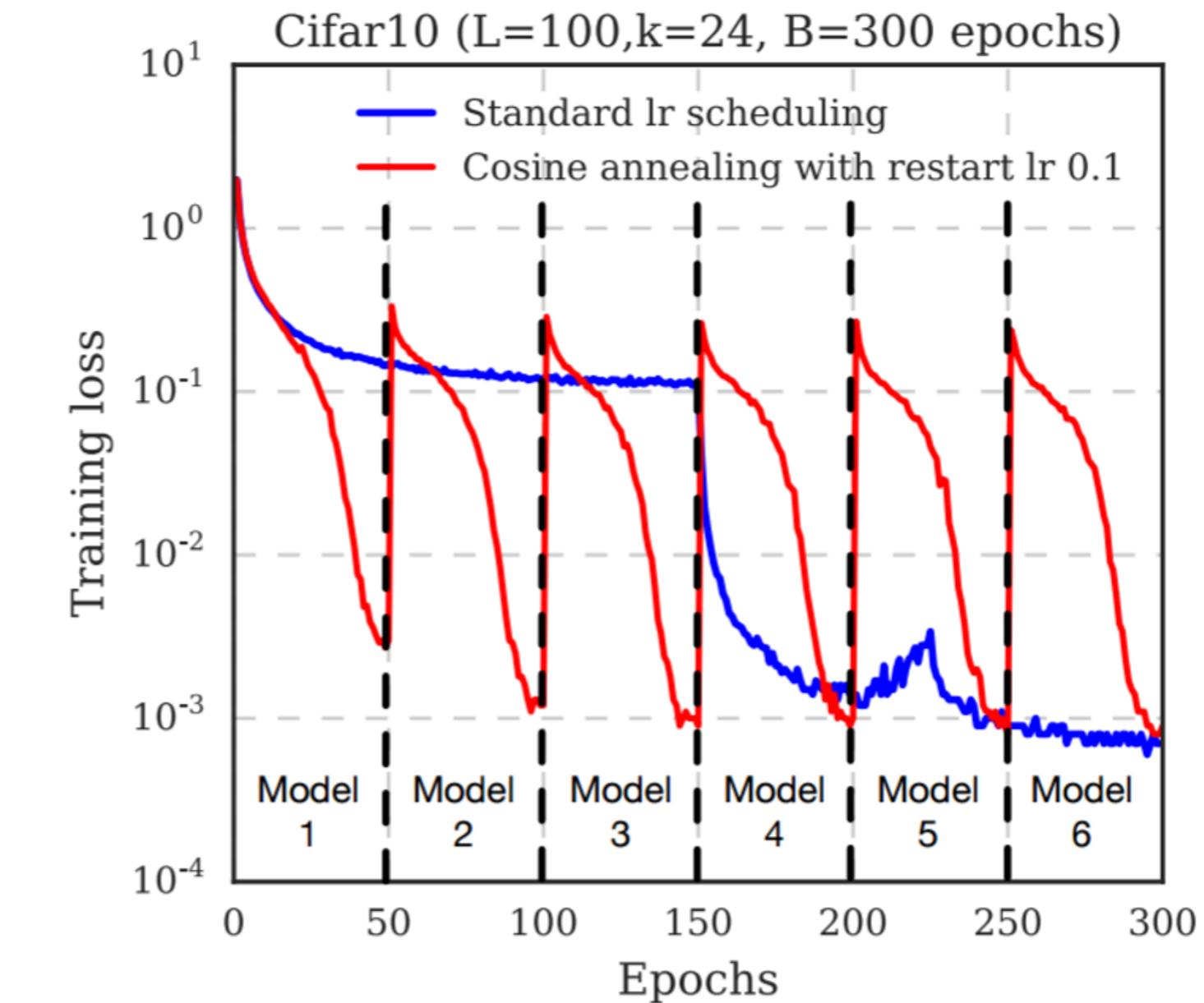
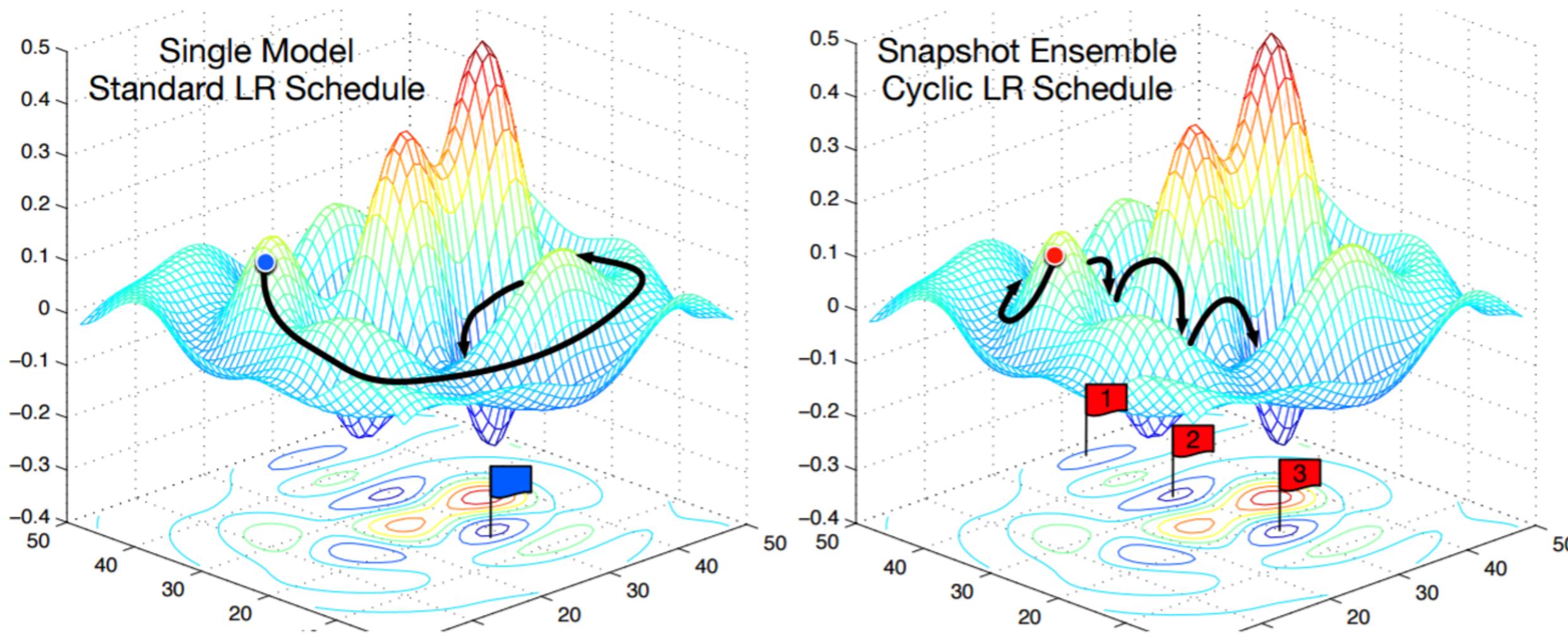
- 1. Train multiple independent models**
- 2. At test time average their results:**

(Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance

Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!



Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016

Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017

Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Cyclic learning rate schedules can make this work even better!

Model Ensembles: Tips and Tricks

Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

```
while True:  
    data_batch = dataset.sample_data_batch()  
    loss = network.forward(data_batch)  
    dx = network.backward()  
    x += - learning_rate * dx  
    x_test = 0.995*x_test + 0.005*x # use for test set
```

Polyak and Juditsky, "Acceleration of stochastic approximation by averaging", SIAM Journal on Control and Optimization, 1992.

Karras et al, "Progressive Growing of GANs for Improved Quality, Stability, and Variation", ICLR 2018

Brock et al, "Large Scale GAN Training for High Fidelity Natural Image Synthesis", ICLR 201

Transfer Learning

“You need a lot of data if you want to
train / use CNNs”

Transfer Learning

“You need a lot of data if you want to
train / use CNNs”

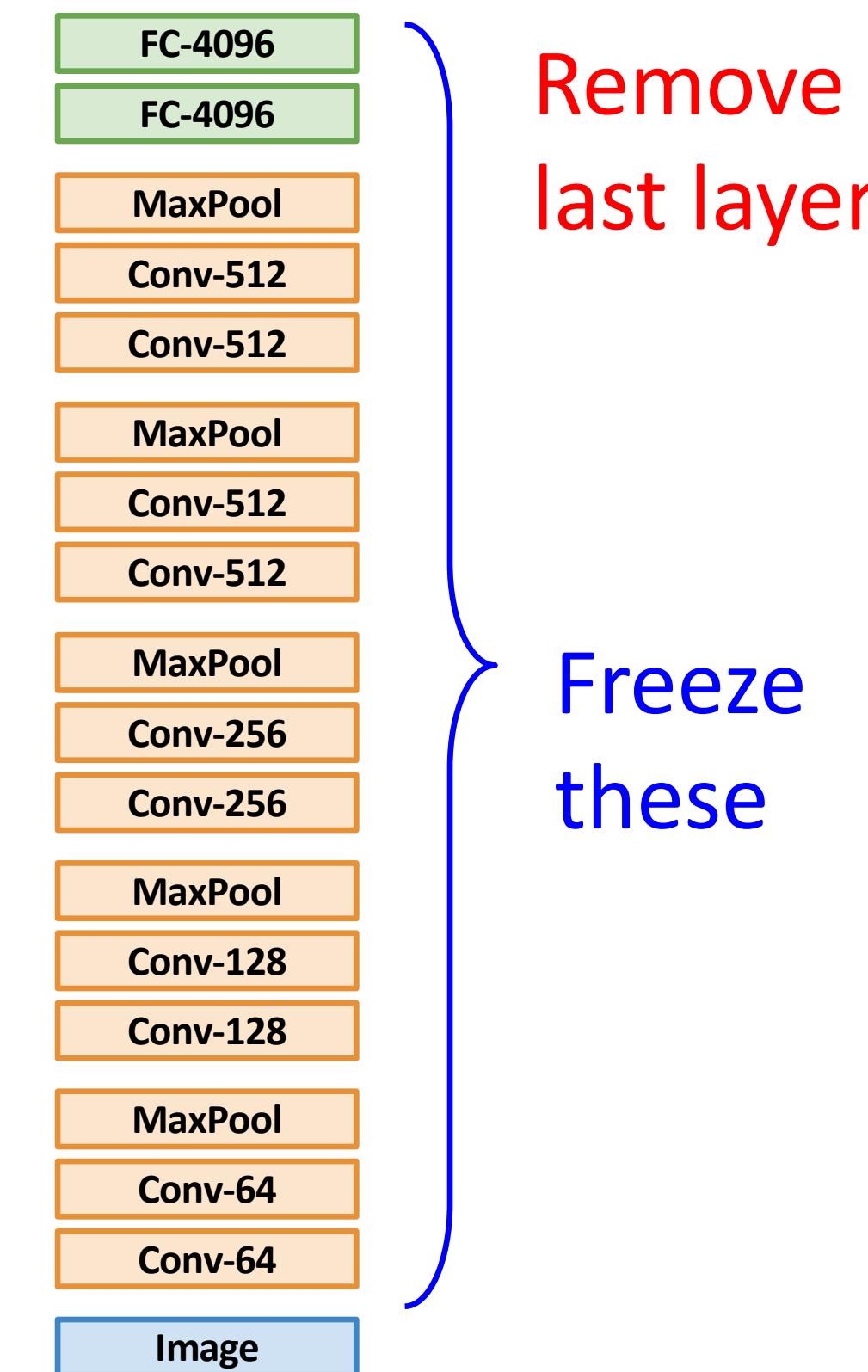
BUSTED

Transfer Learning with CNNs

1. Train on ImageNet

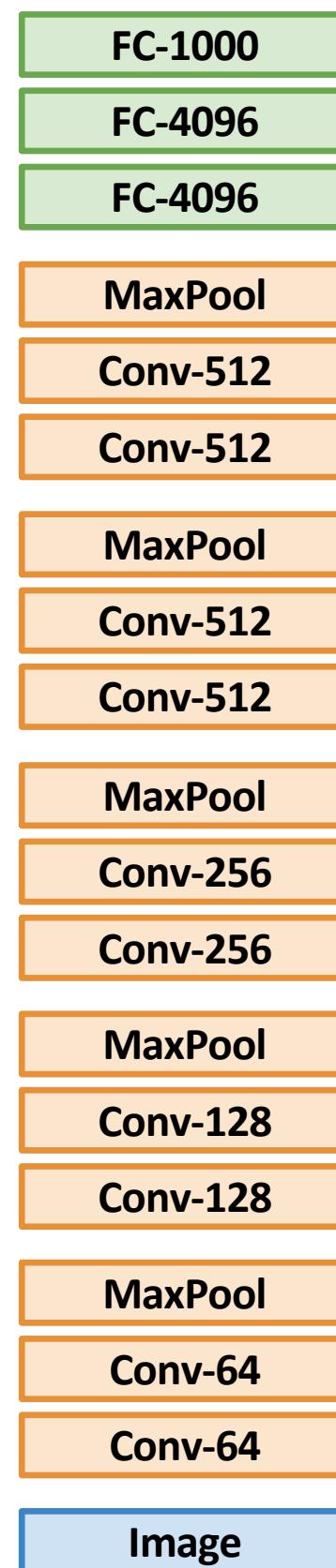


2. Use CNN as a feature extractor

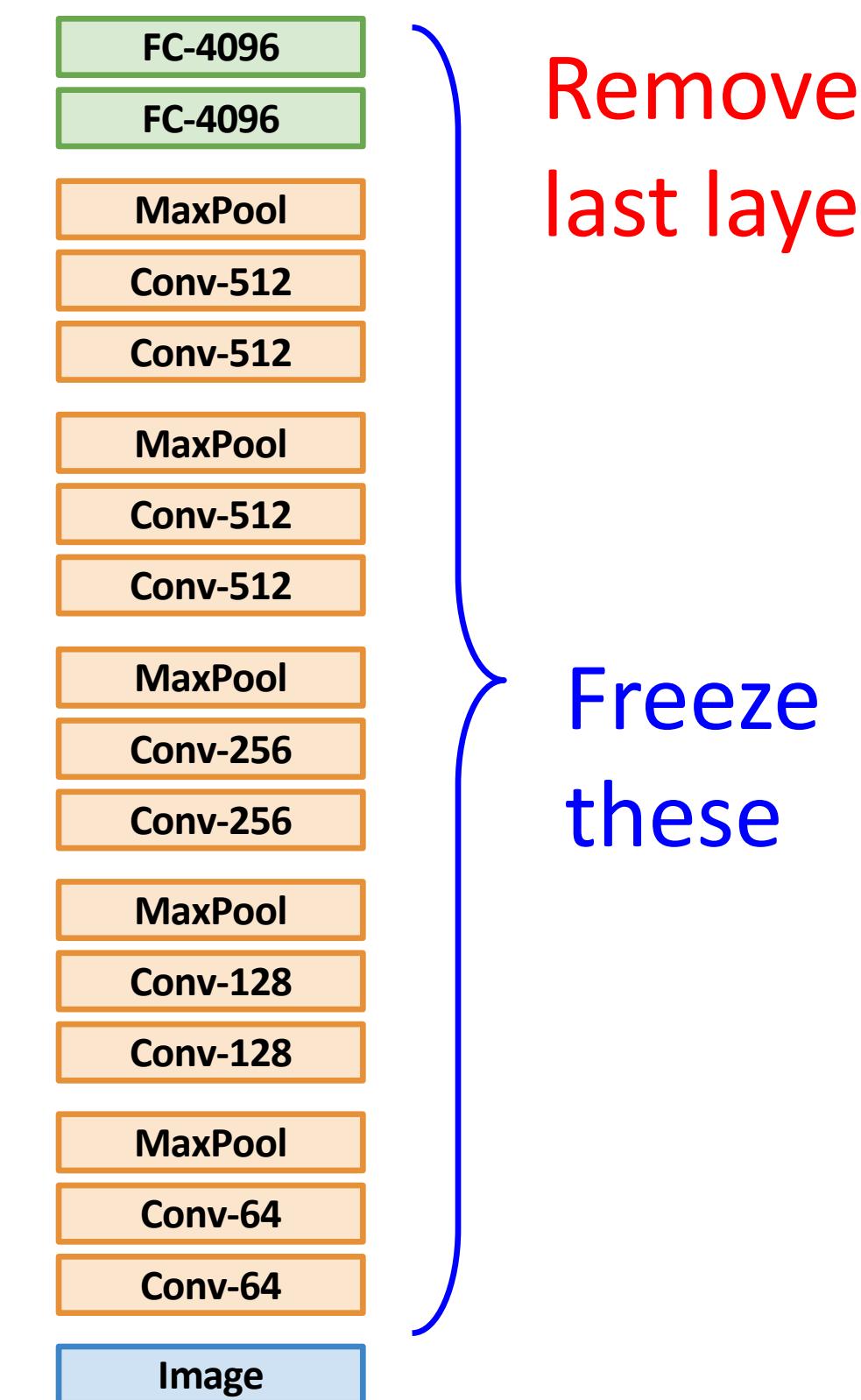


Transfer Learning with CNNs

1. Train on ImageNet



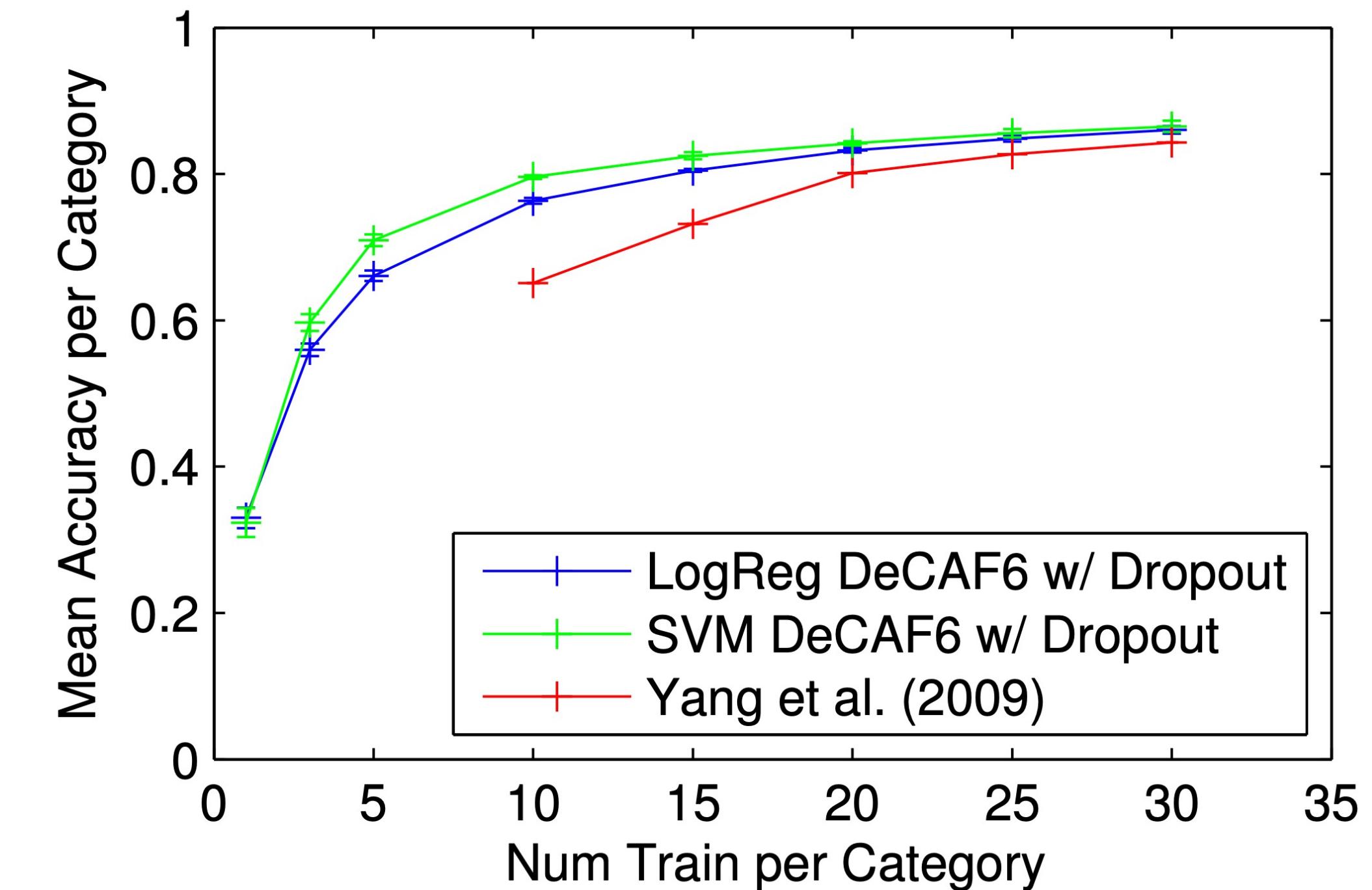
2. Use CNN as a feature extractor



Remove
last layer

Freeze
these

Classification on Caltech-101

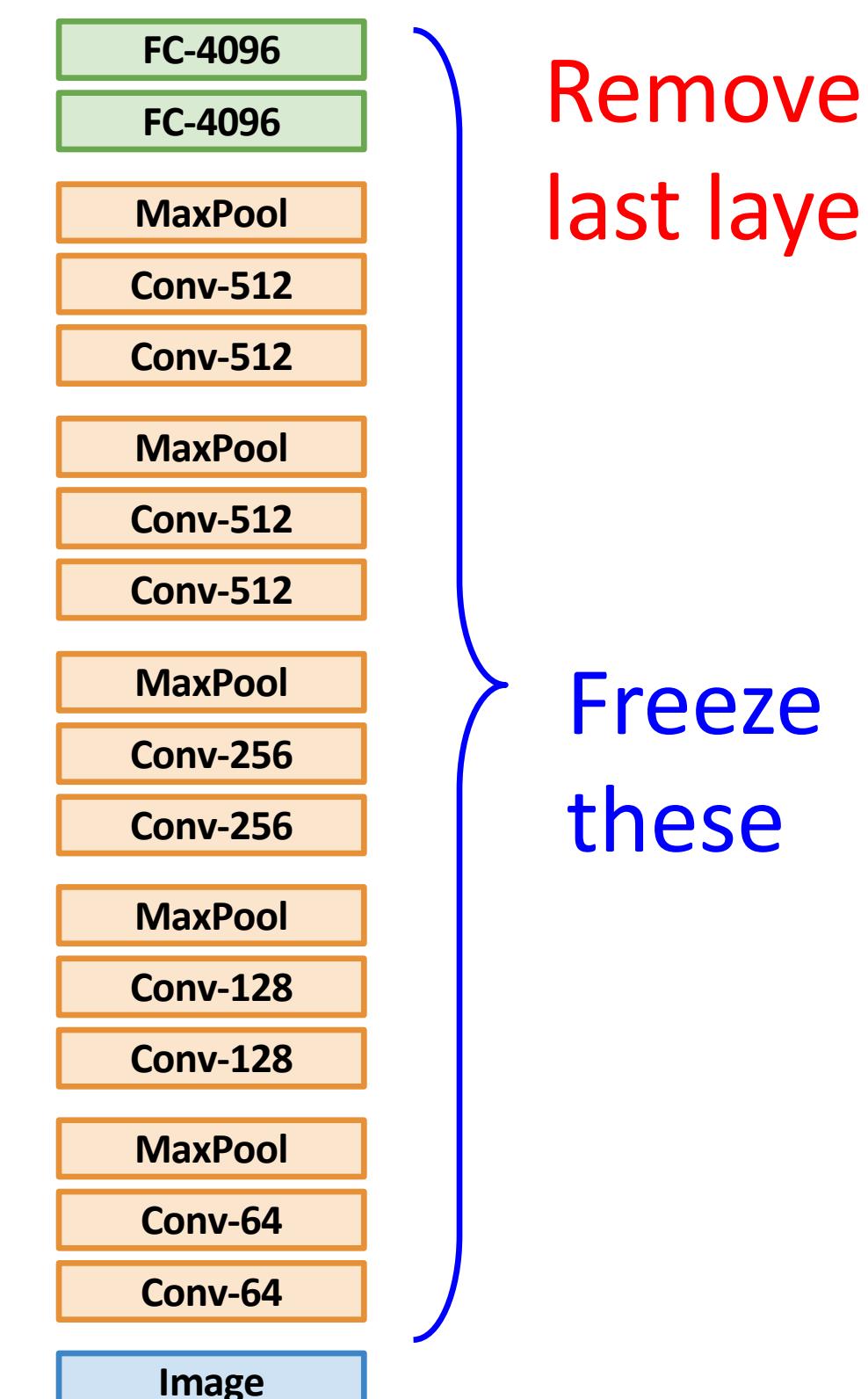


Transfer Learning with CNNs

1. Train on ImageNet



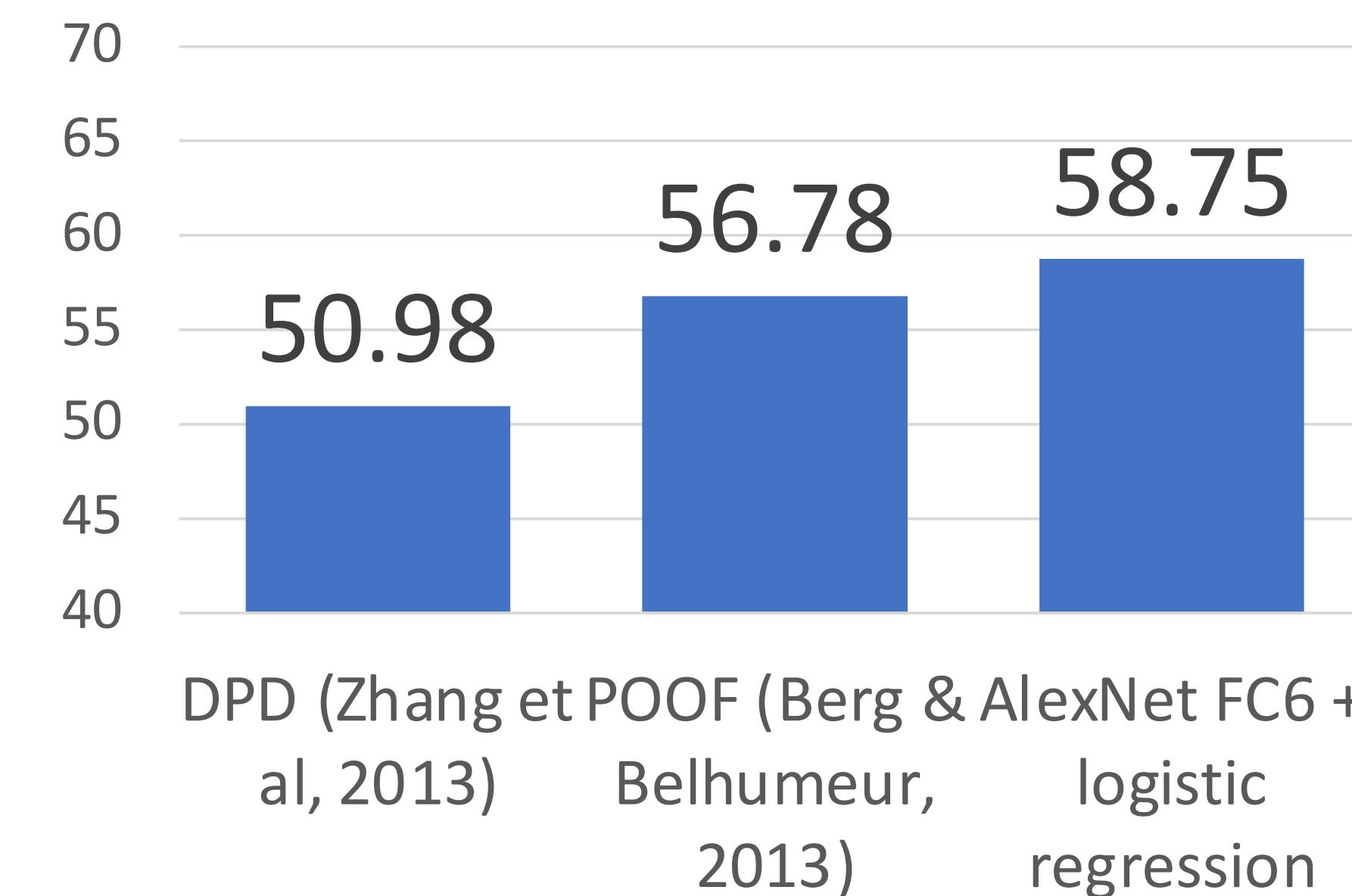
2. Use CNN as a feature extractor



Remove
last layer

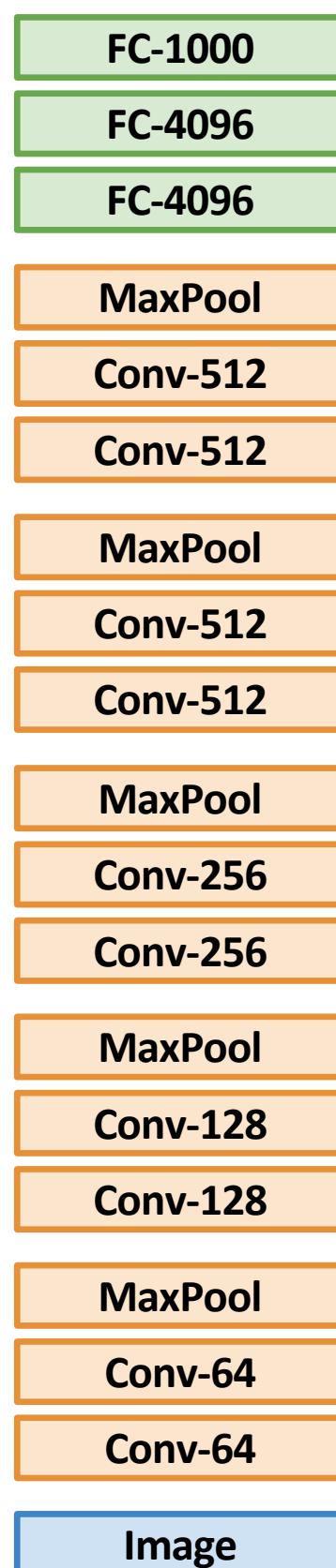
Freeze these

Bird Classification on Caltech-UCSD

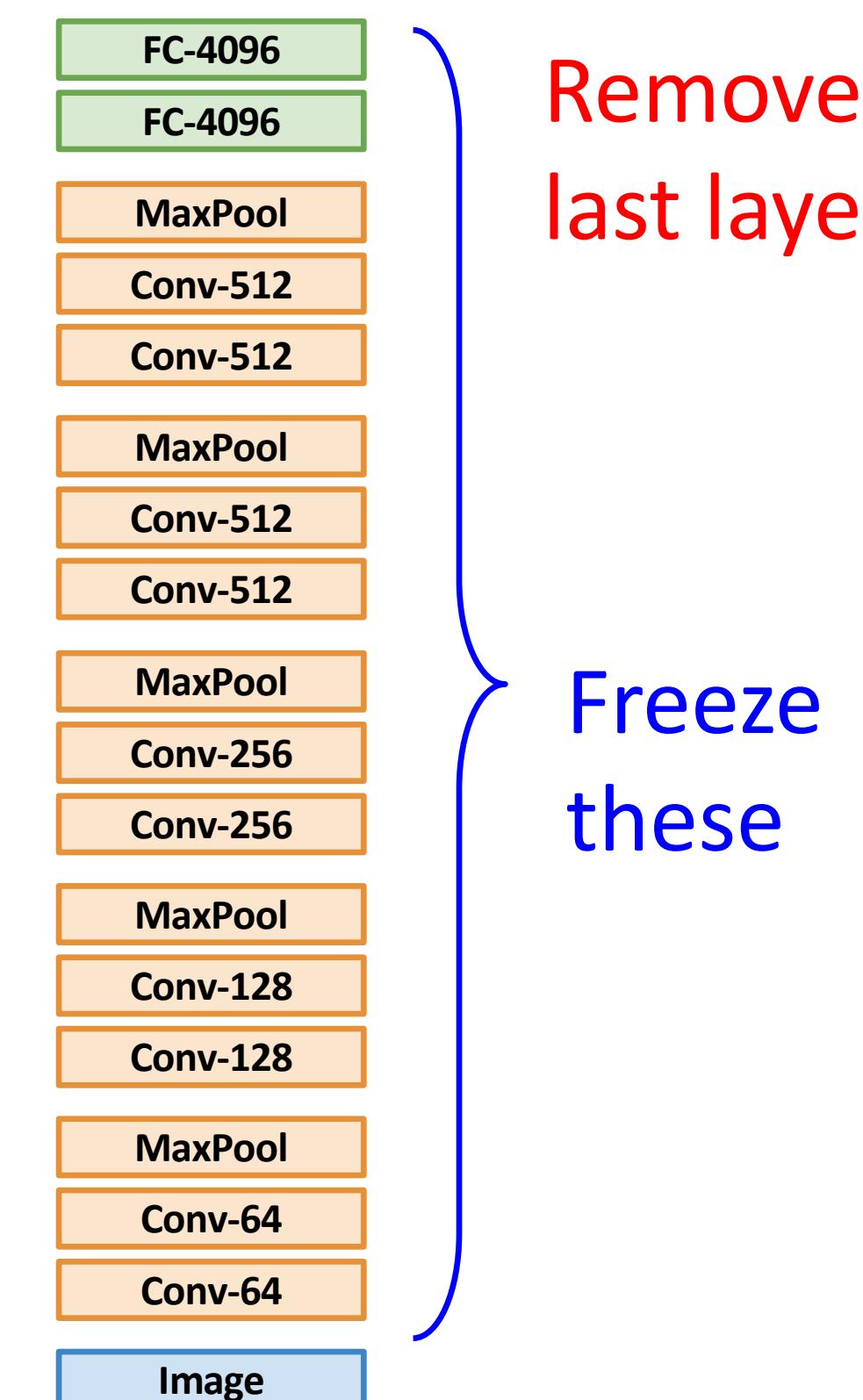


Transfer Learning with CNNs

1. Train on ImageNet



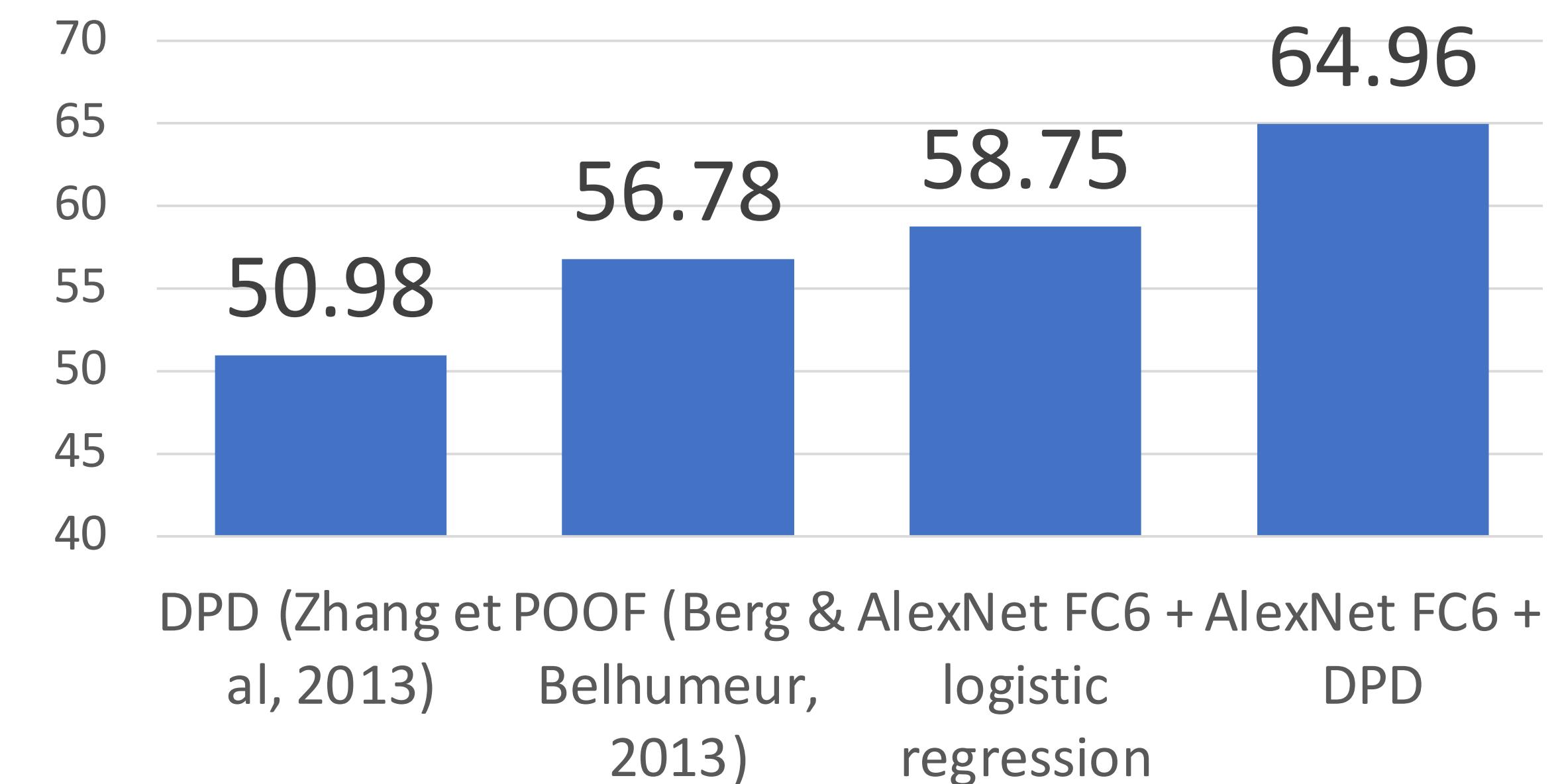
2. Use CNN as a feature extractor



Remove
last layer

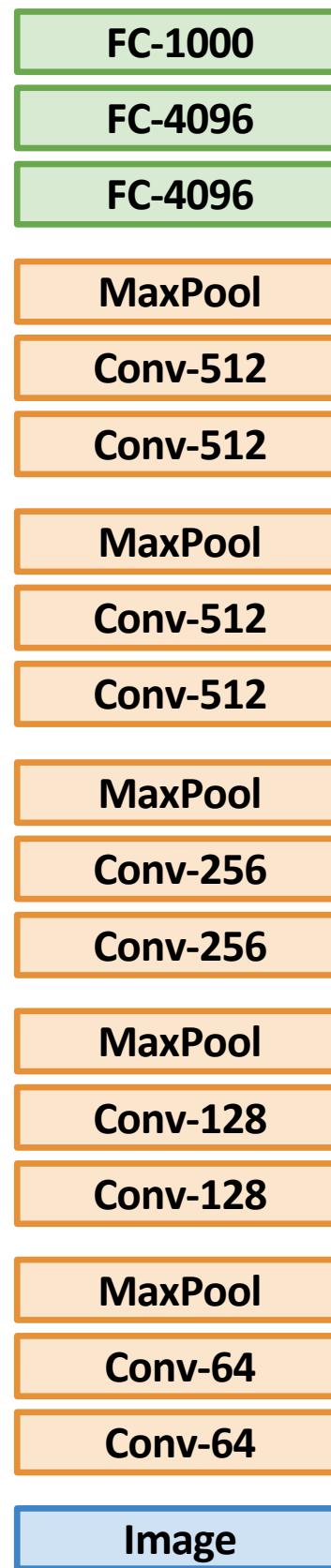
→ Freeze these

Bird Classification on Caltech-UCSD



Transfer Learning with CNNs

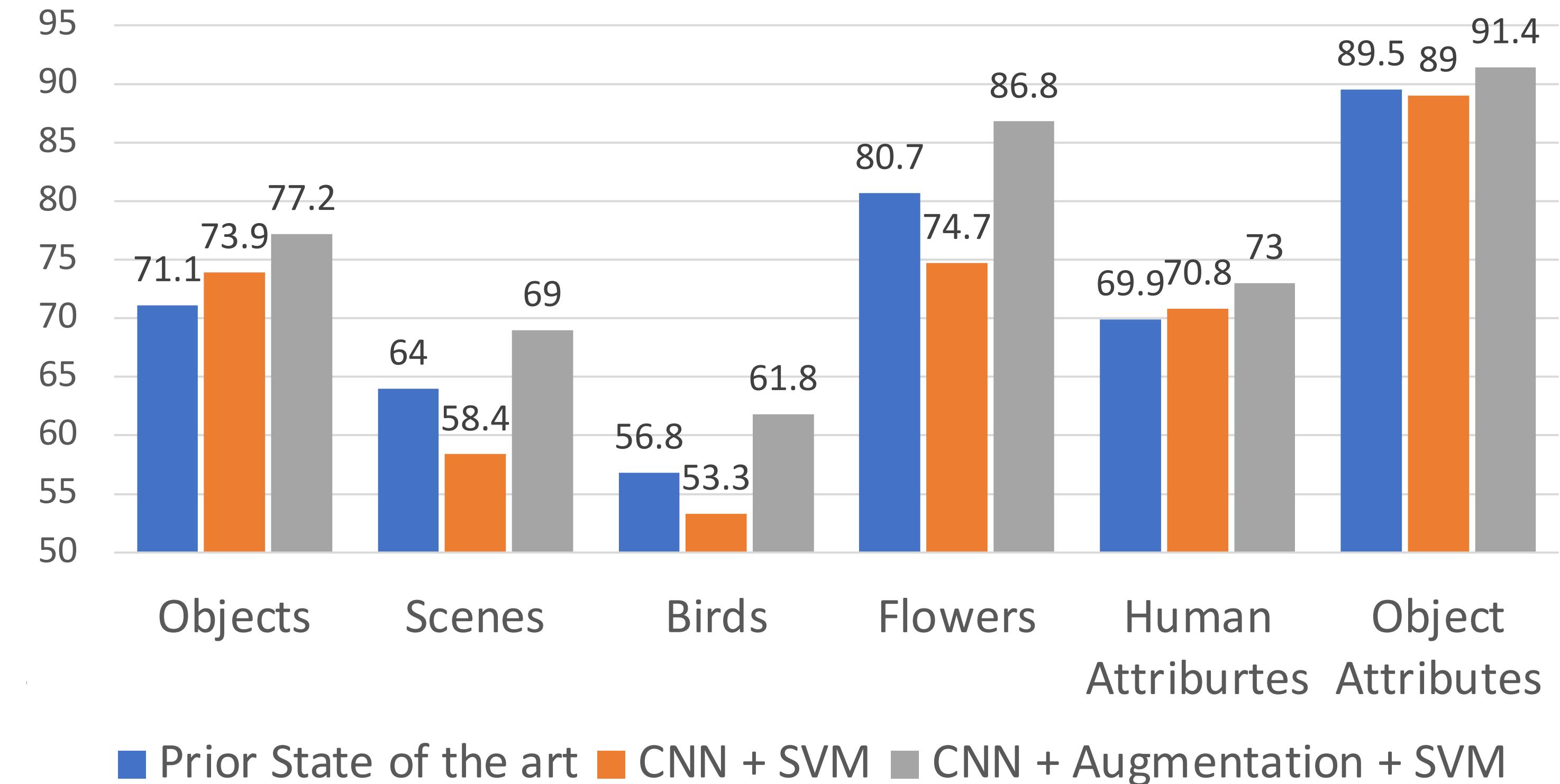
1. Train on ImageNet



2. Use CNN as a feature extractor



Image Classification



Transfer Learning with CNNs

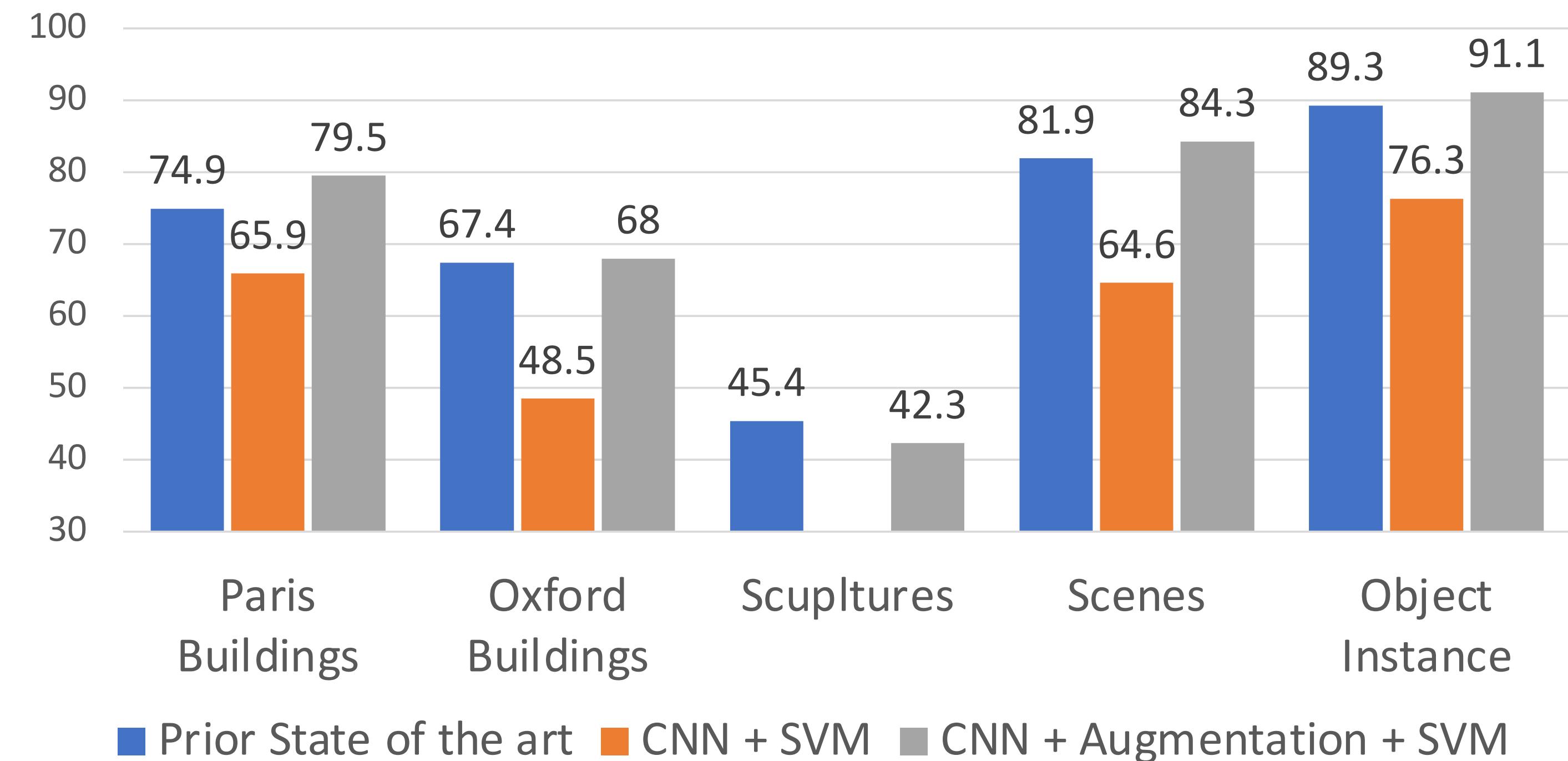
1. Train on ImageNet



2. Use CNN as a feature extractor



Image Retrieval: Nearest-Neighbor

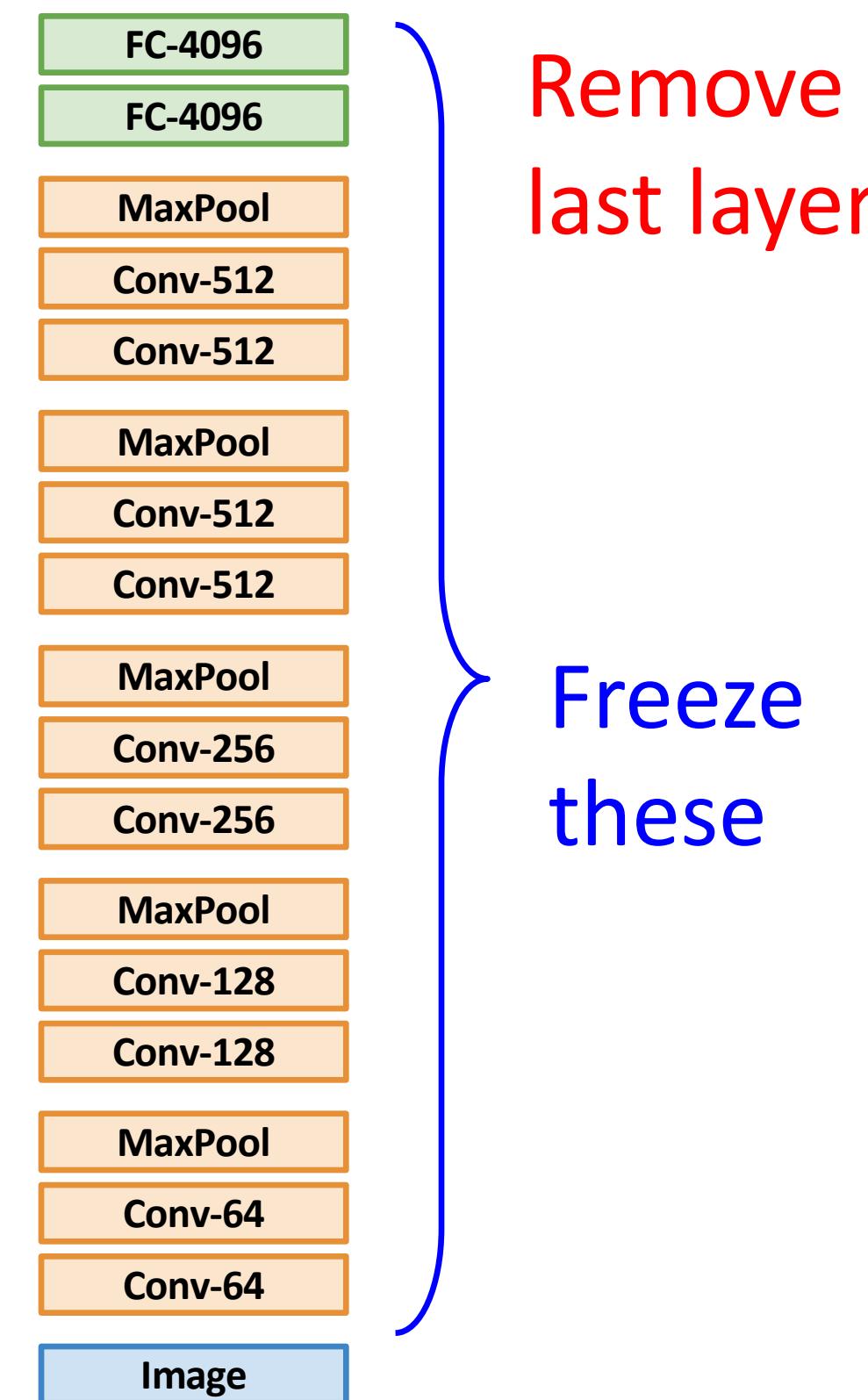


Transfer Learning with CNNs

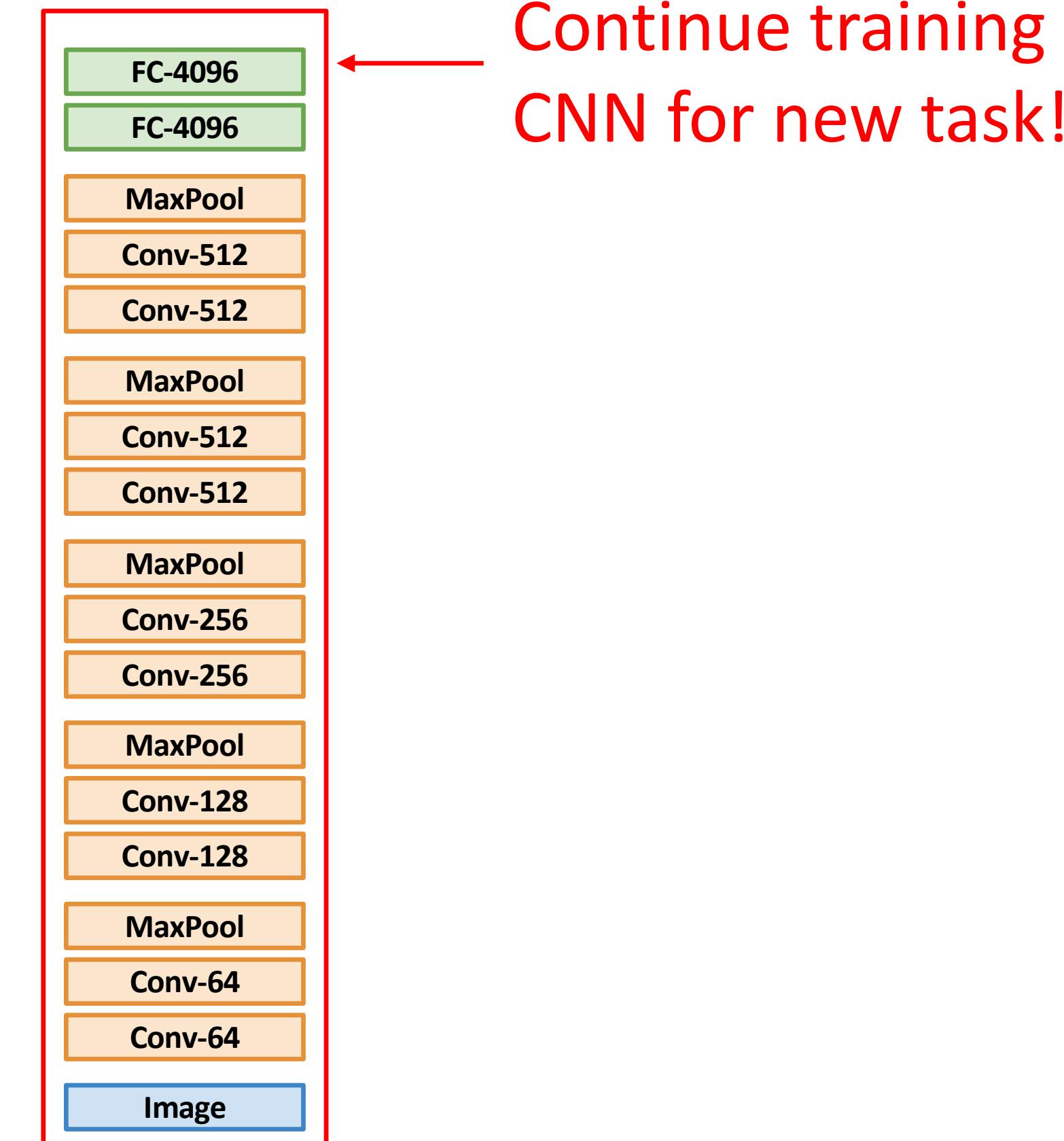
1. Train on ImageNet



2. Use CNN as a feature extractor



3. Bigger dataset: Fine-Tuning

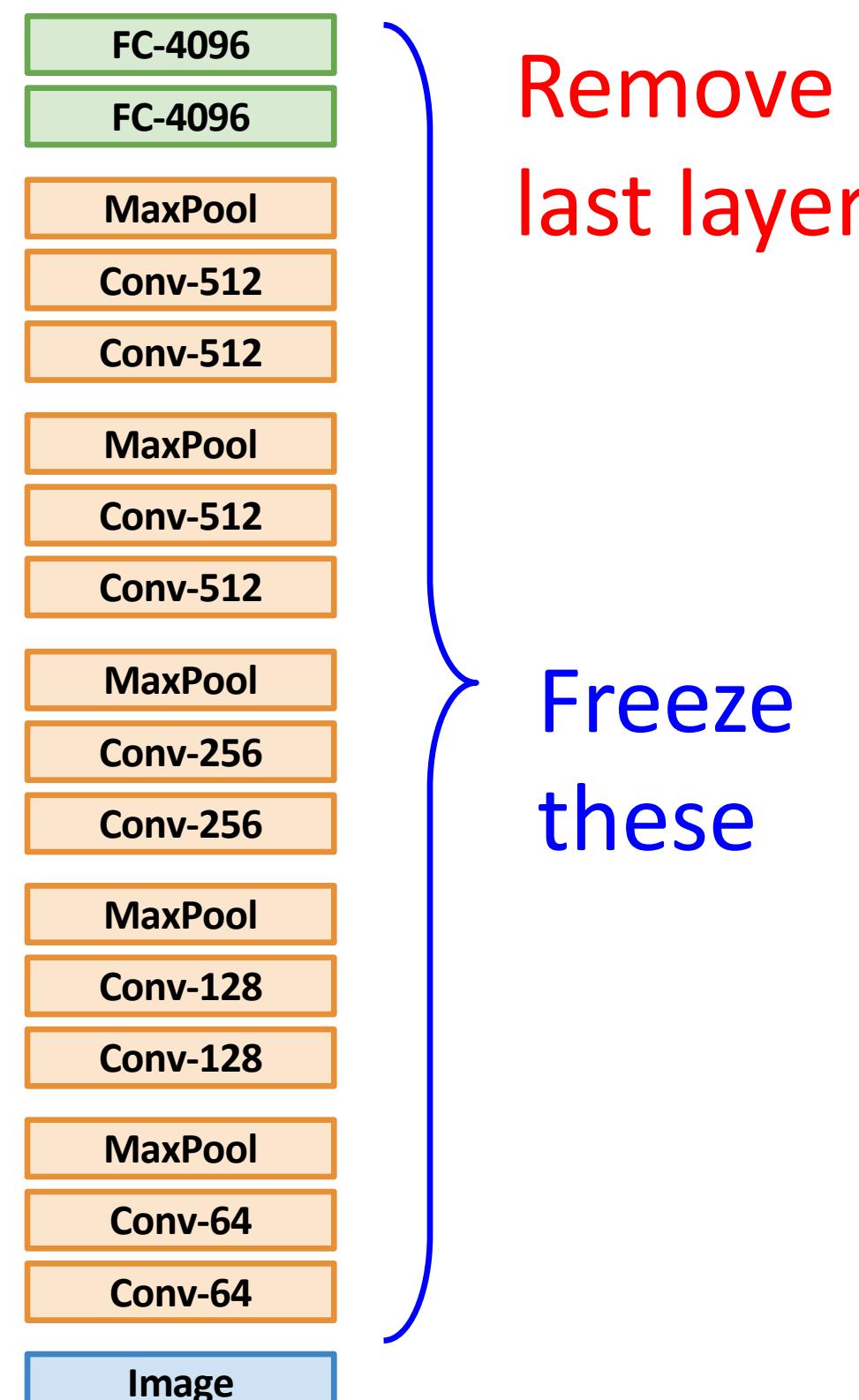


Transfer Learning with CNNs

1. Train on ImageNet

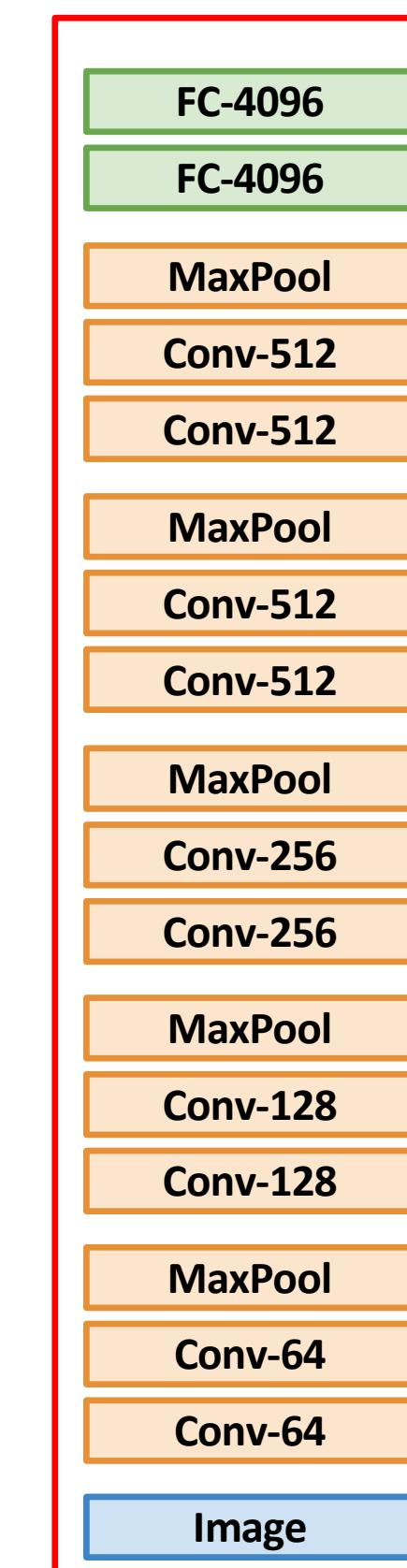


2. Use CNN as a feature extractor



3. Bigger dataset

Fine-Tuning



- Continue training CNN for new task!

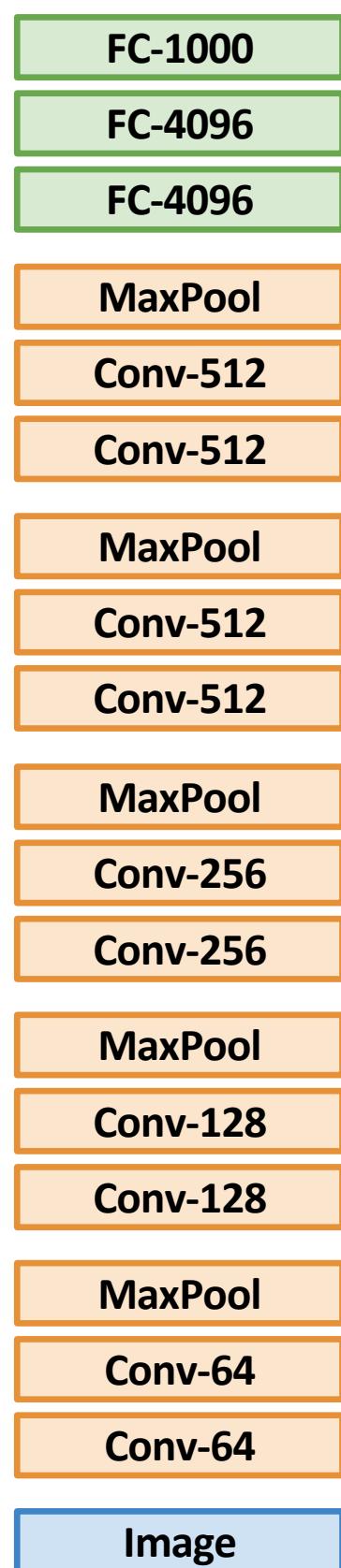
Some tricks

- Train with feature extraction first before fine-tuning
 - Lower the learning rate: use $\sim 1/10$ of LR used in original training
 - Sometimes freeze lower layers to save computation
 - Train with BatchNorm in “test” mode

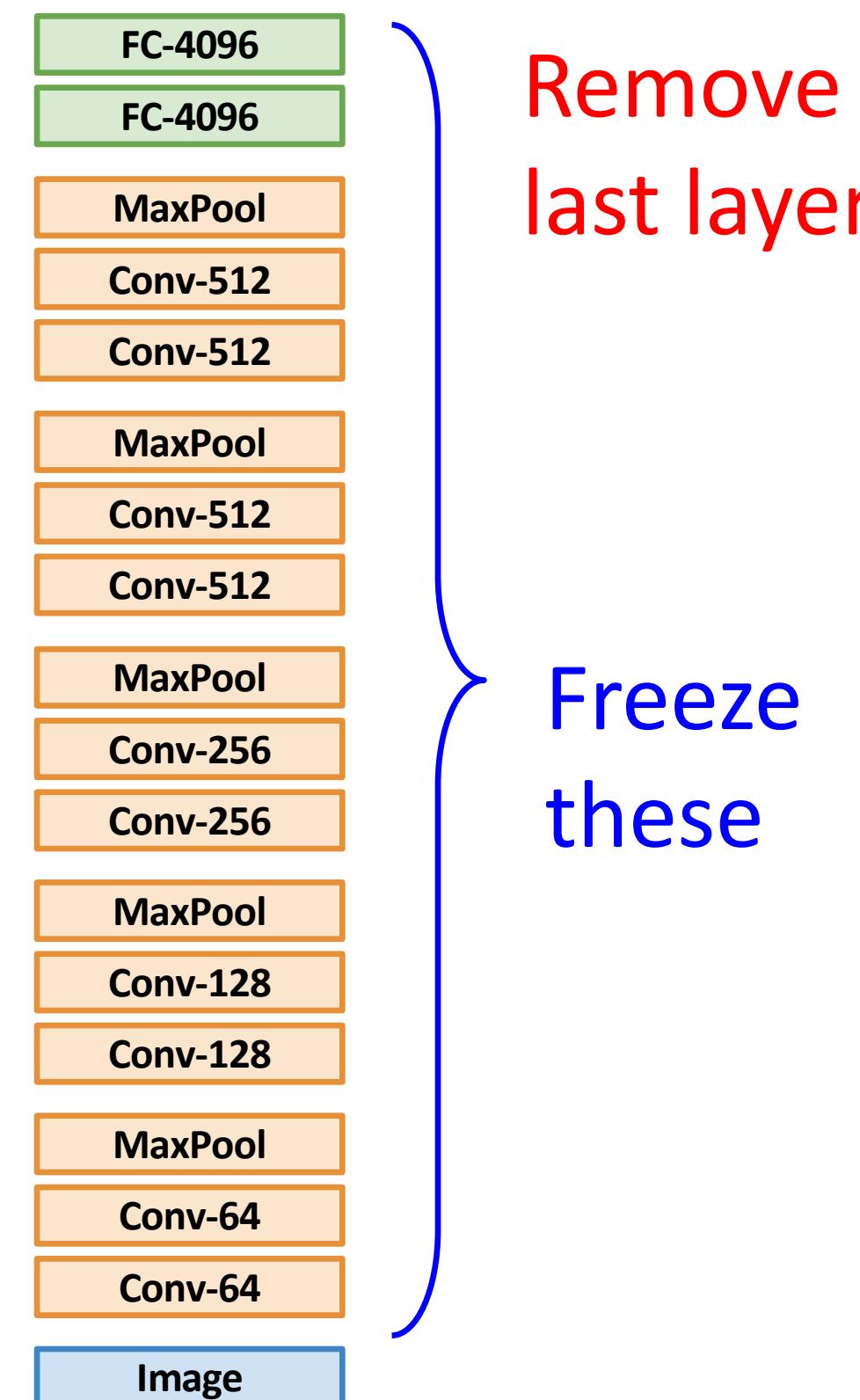


Transfer Learning with CNNs

1. Train on ImageNet



2. Use CNN as a feature extractor

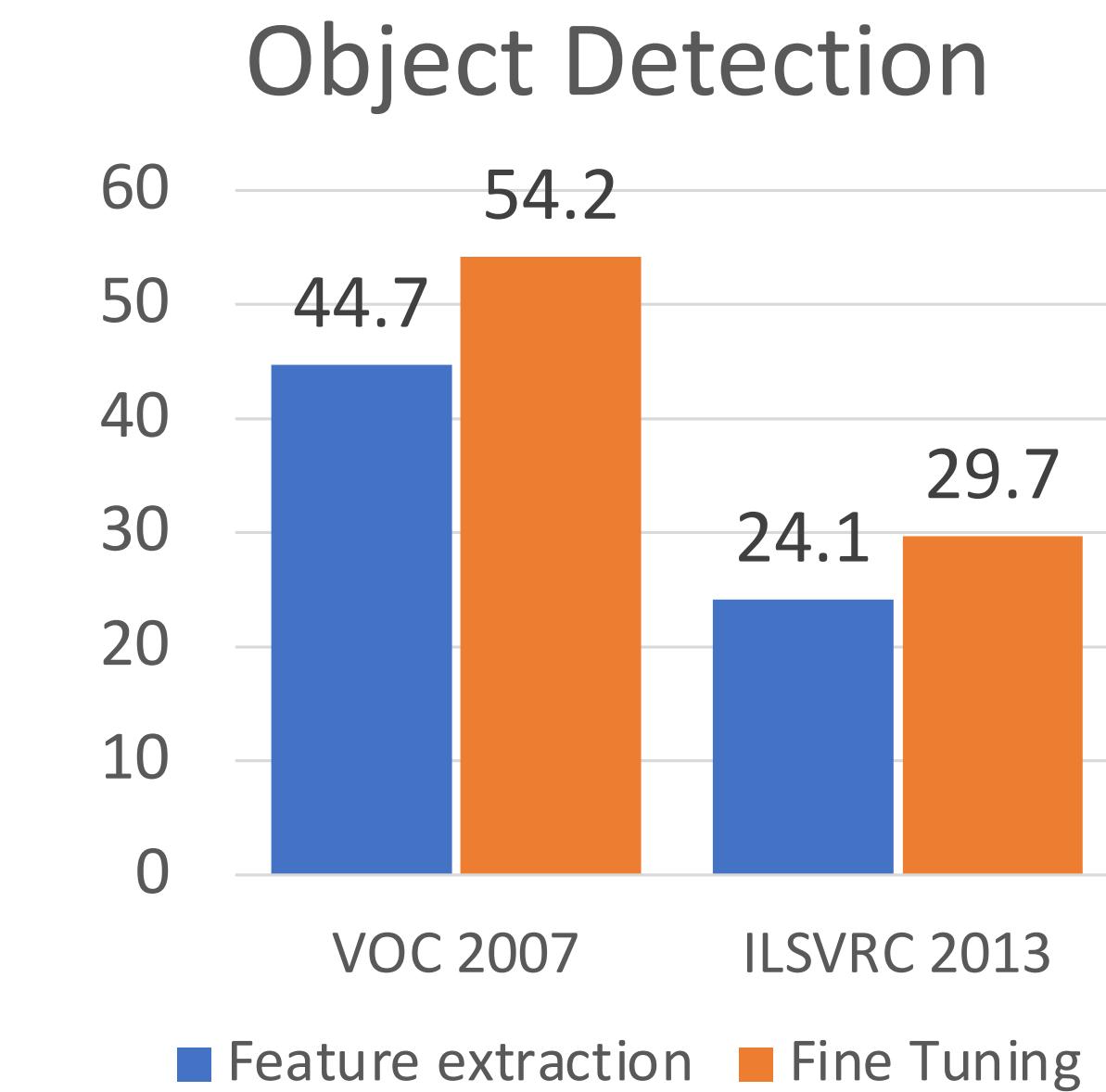


3. Bigger dataset

Fine-Tuning

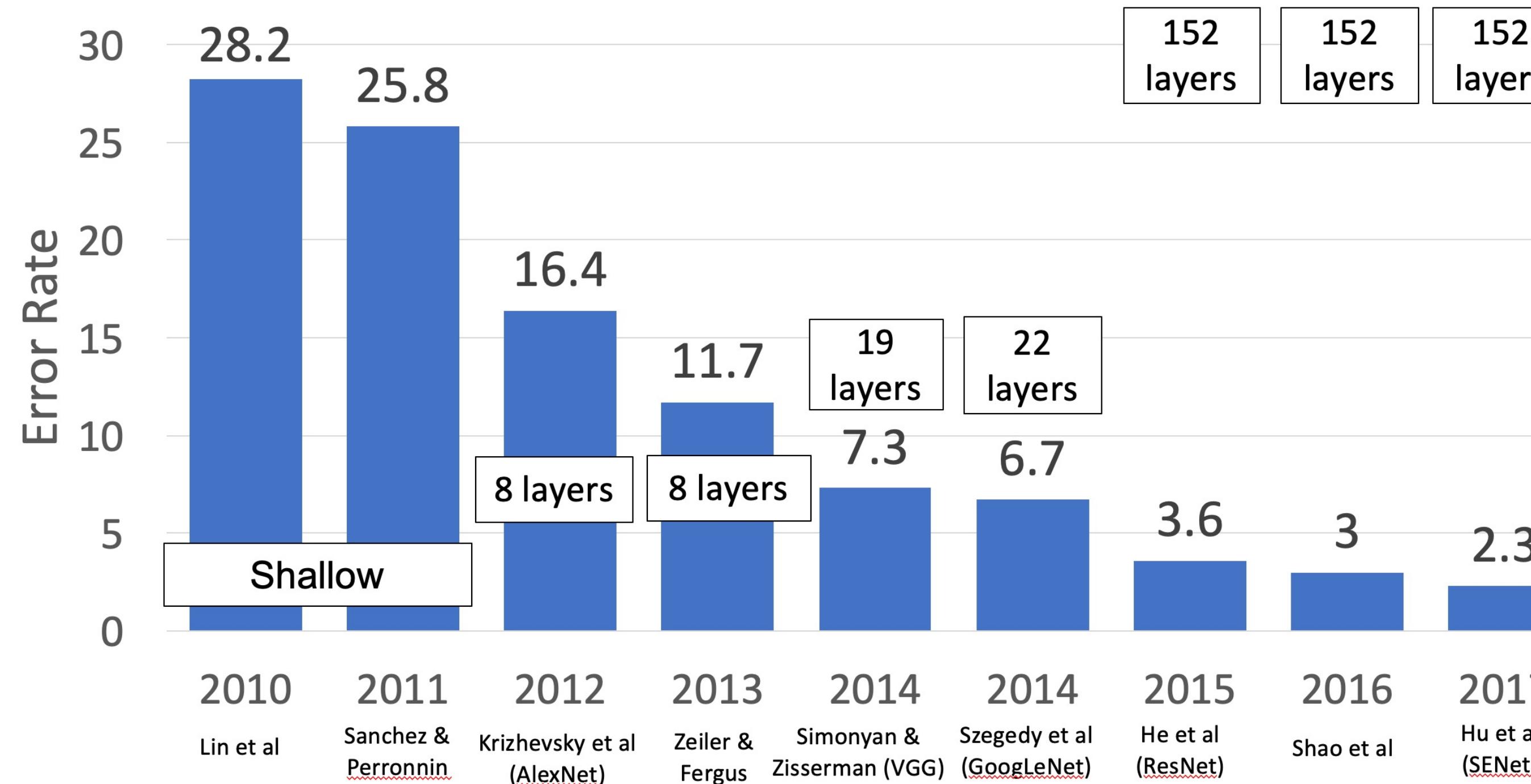


— Continue training CNN for new task!



Transfer Learning with CNNs: Architecture Matters!

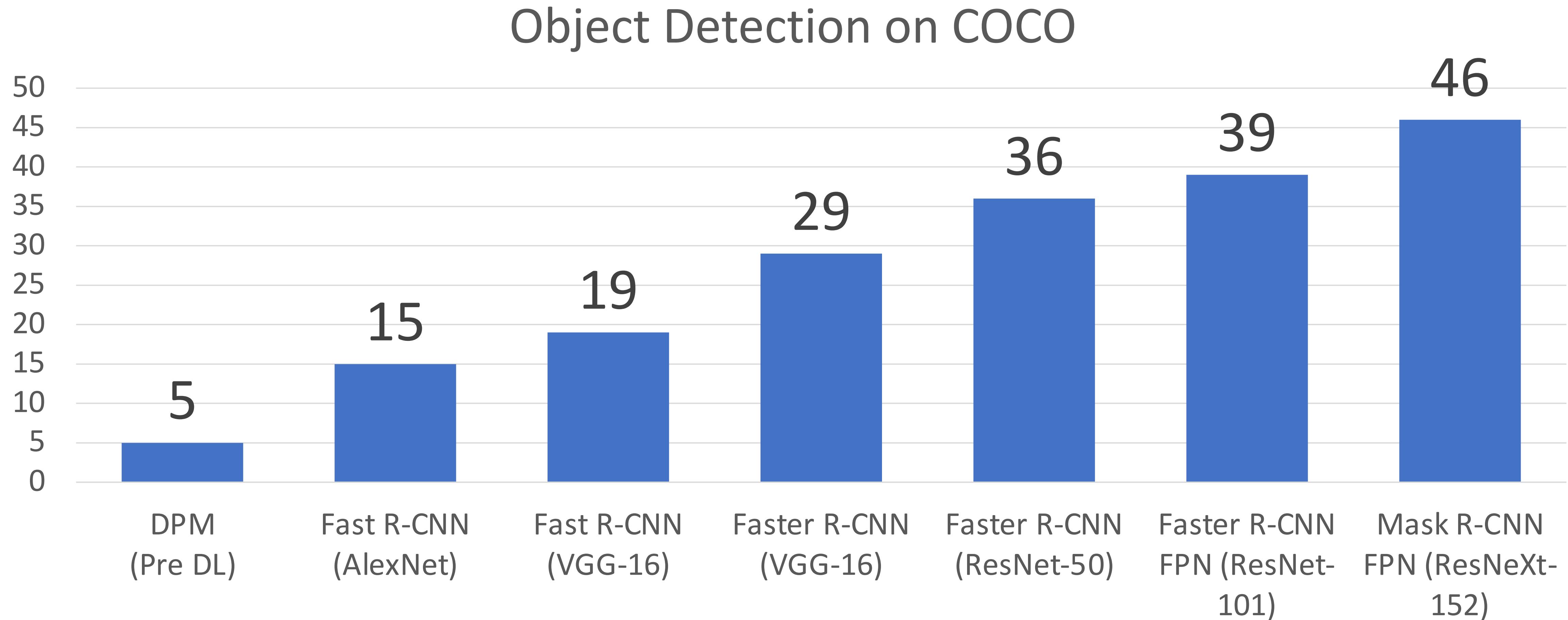
ImageNet Classification Challenge



Improvements in CNN architectures lead to improvements in many downstream tasks thanks to transfer learning!



Transfer Learning with CNNs: Architecture Matters!



Transfer Learning with CNNs



More specific

More generic

	Dataset similar to ImageNet	Dataset very different from ImageNet
Very little data (10s to 100s)	?	?
Quite a lot of data (100s to 1000s)	?	?

Transfer Learning with CNNs



More specific

More generic

	Dataset similar to ImageNet	Dataset very different from ImageNet
Very little data (10s to 100s)	Use Linear Classifier on top layer	?
Quite a lot of data (100s to 1000s)	Finetune a few layers	?

Transfer Learning with CNNs



More specific

More generic

	Dataset similar to ImageNet	Dataset very different from ImageNet
Very little data (10s to 100s)	Use Linear Classifier on top layer	?
Quite a lot of data (100s to 1000s)	Finetune a few layers	Finetune a larger number of layers

Transfer Learning with CNNs



More specific

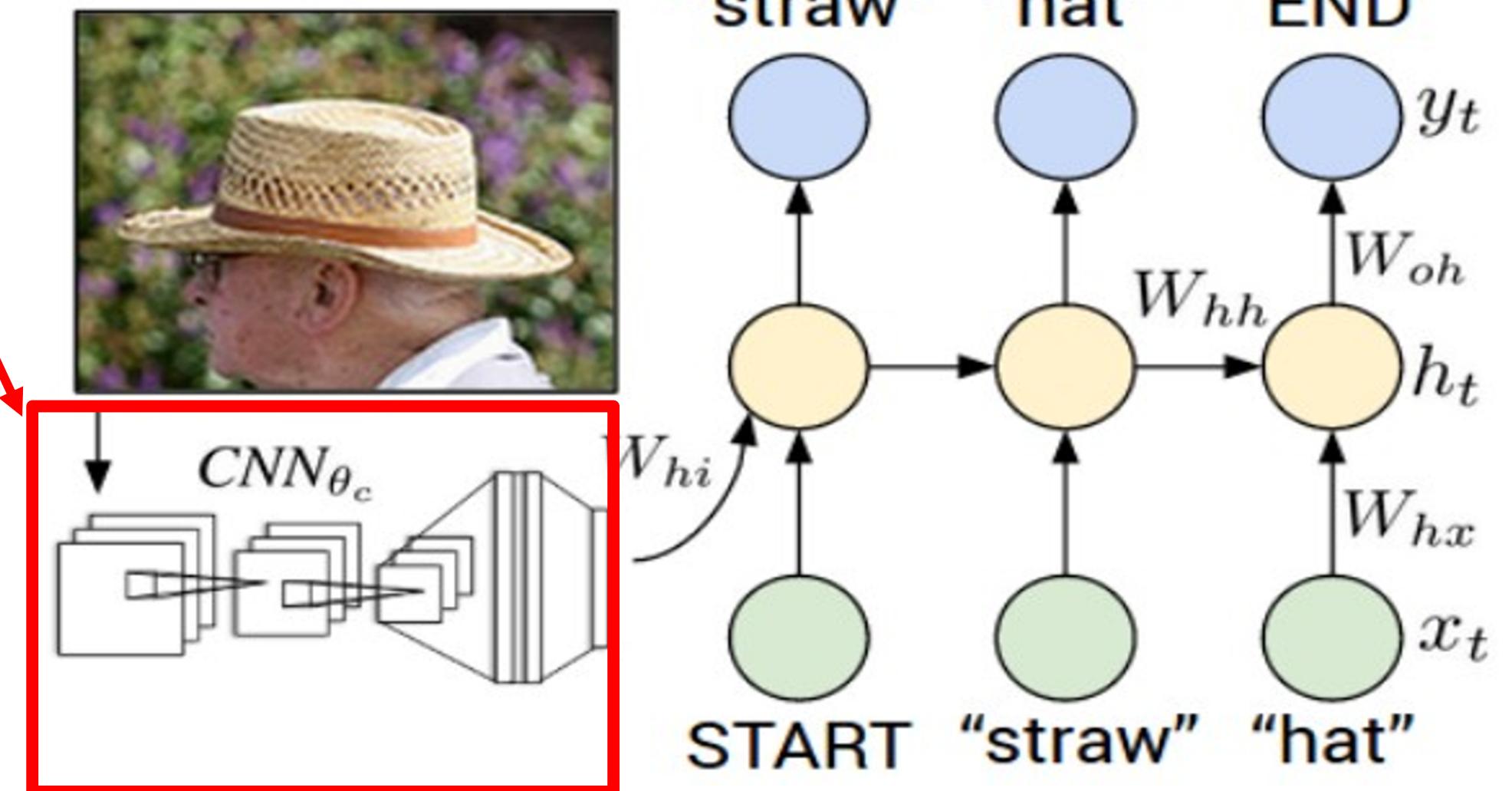
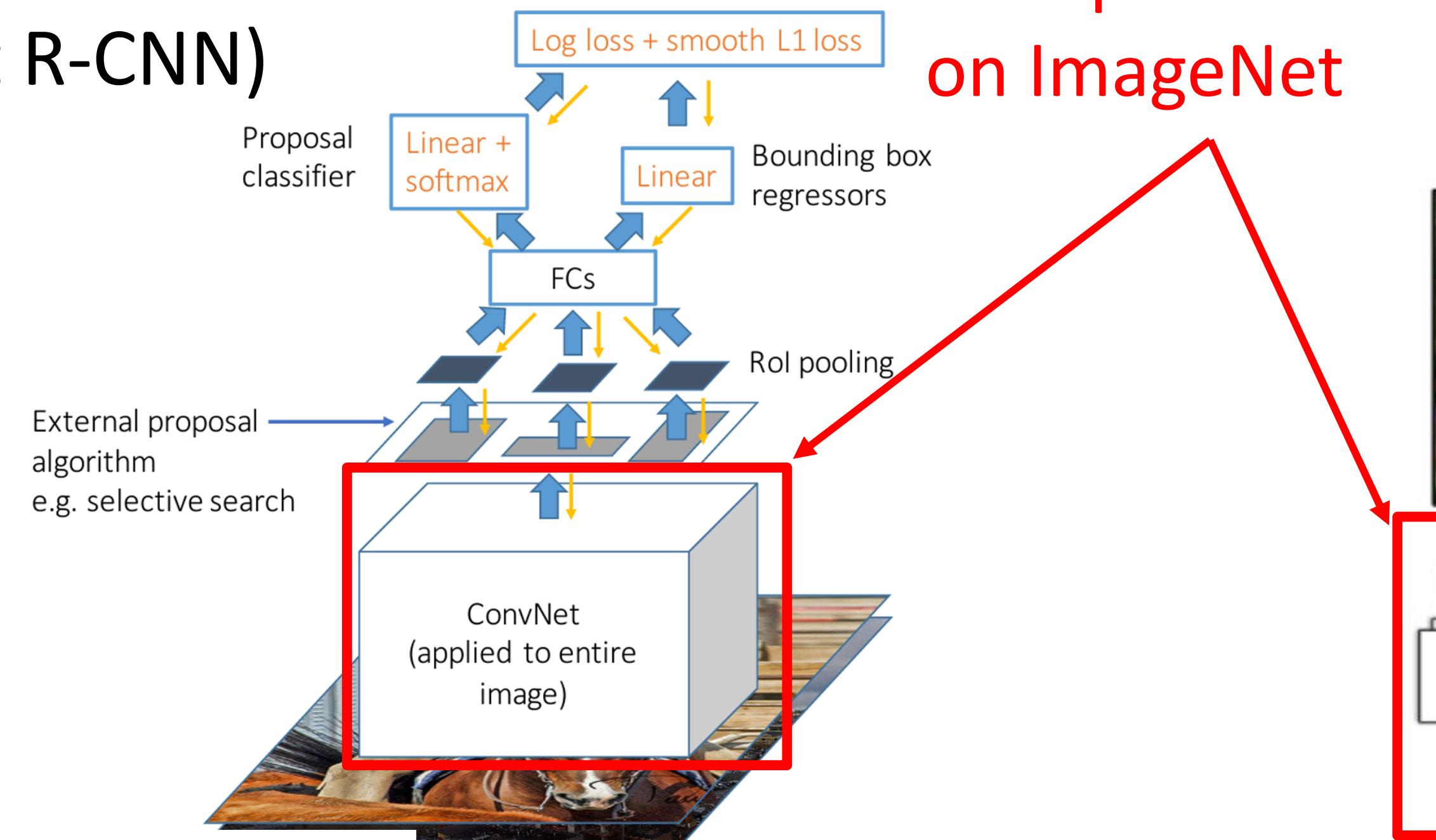
More generic

	Dataset similar to ImageNet	Dataset very different from ImageNet
Very little data (10s to 100s)	Use Linear Classifier on top layer	You're in trouble... Try linear classifier from different stages
Quite a lot of data (100s to 1000s)	Finetune a few layers	Finetune a larger number of layers

Transfer Learning is pervasive!

Its the norm, not the exception

Object Detection (Fast R-CNN)



Girshick, "Fast R-CNN", ICCV 2015

Figure copyright Ross Girshick, 2015. Reproduced with permission.

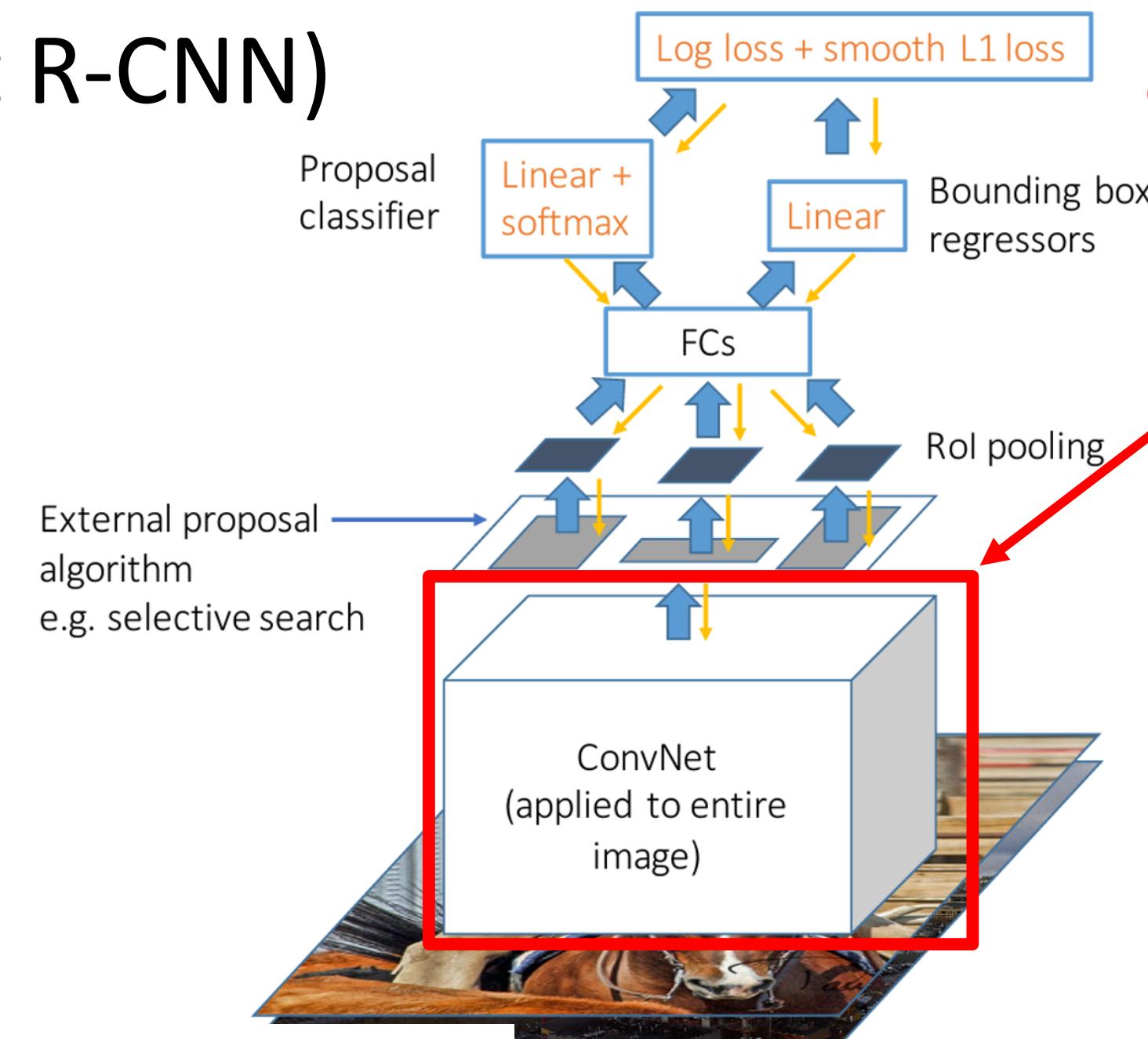


Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

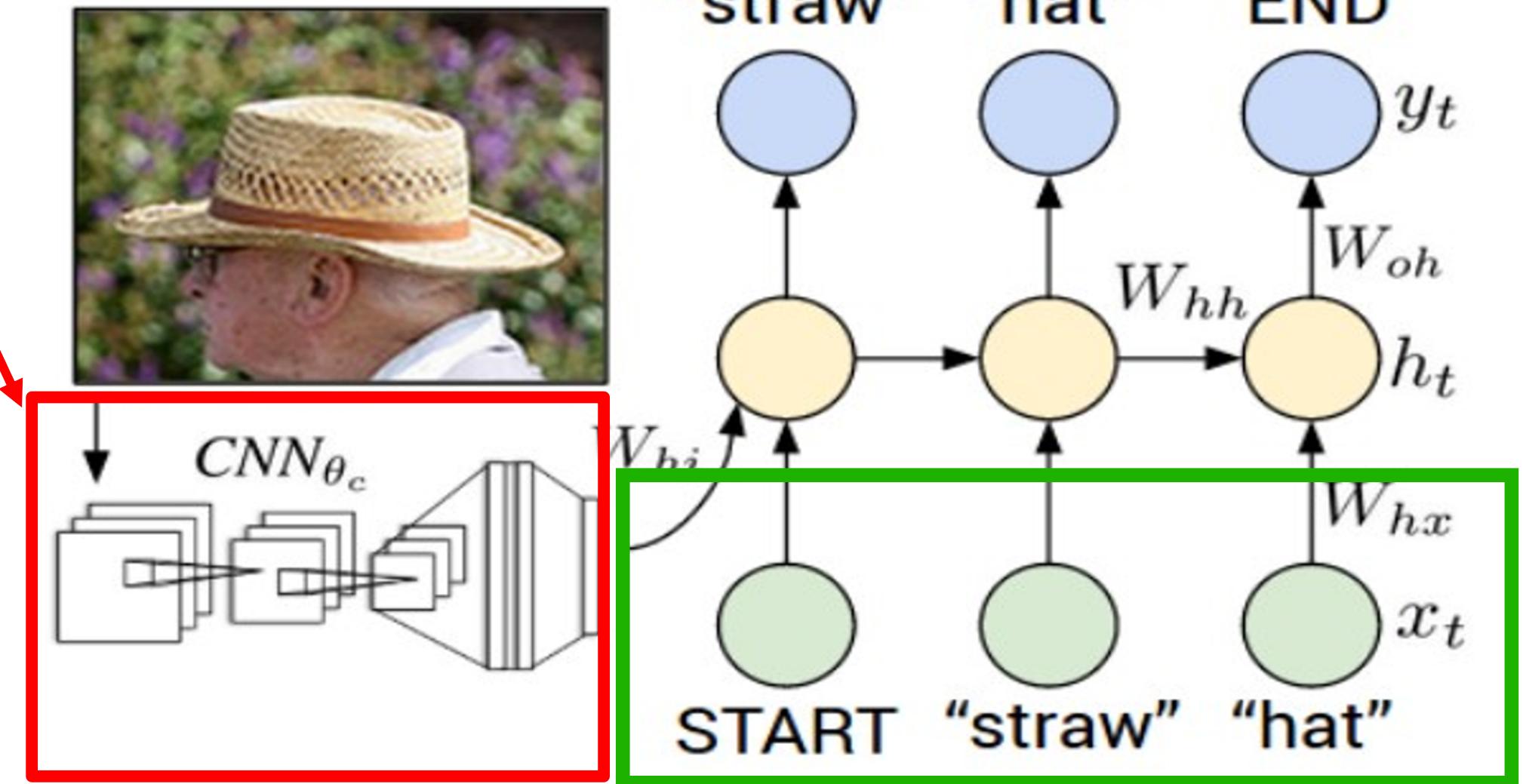
Transfer Learning is pervasive!

Its the norm, not the exception

Object Detection (Fast R-CNN)



CNN pretrained
on ImageNet



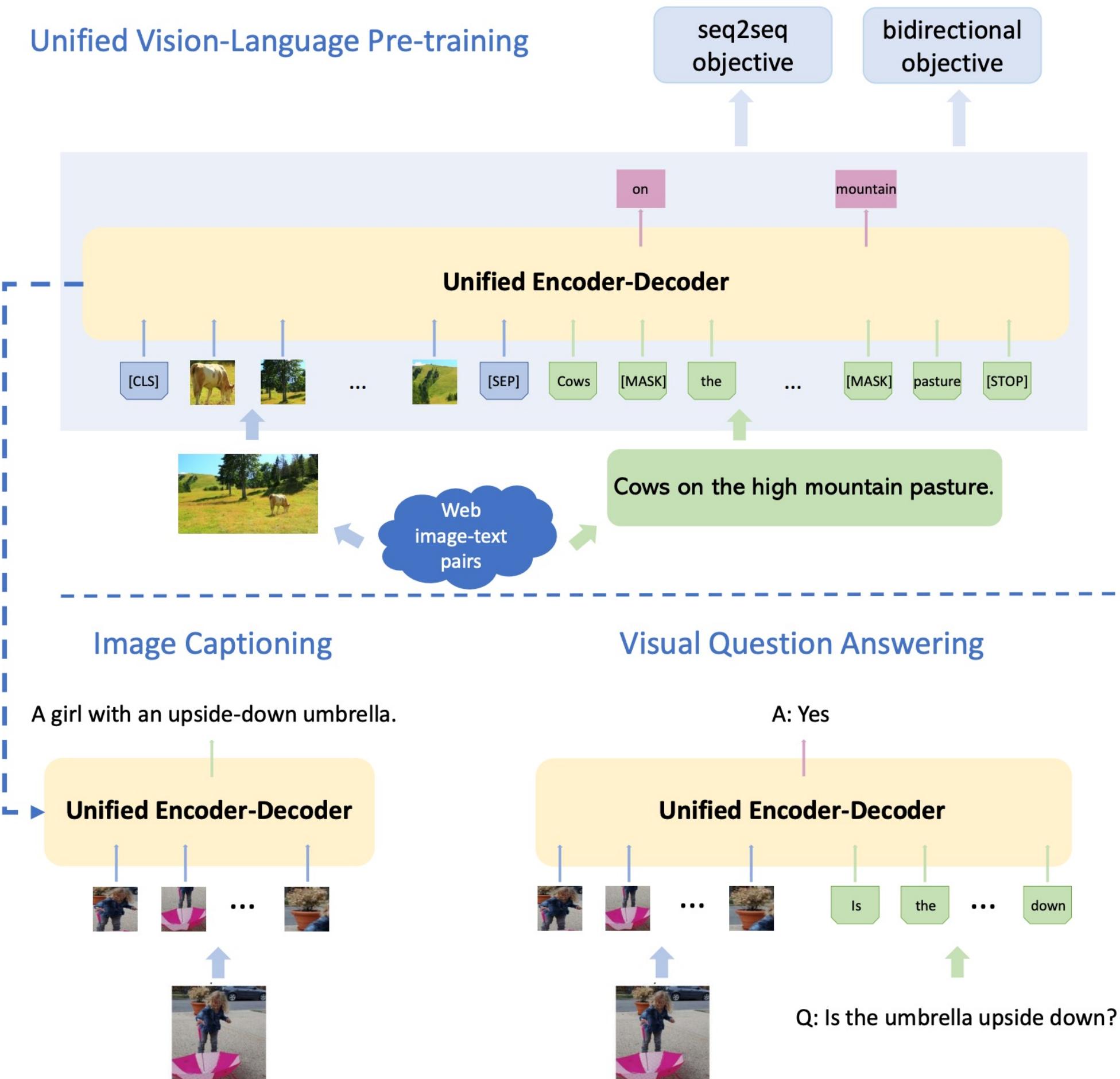
Girshick, “Fast R-CNN”, ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

Word vectors pretrained
with word2vec

Karpathy and Fei-Fei, “Deep Visual-Semantic Alignments for Generating Image Descriptions”, CVPR 2015

Transfer Learning is pervasive!

Its the norm, not the exception



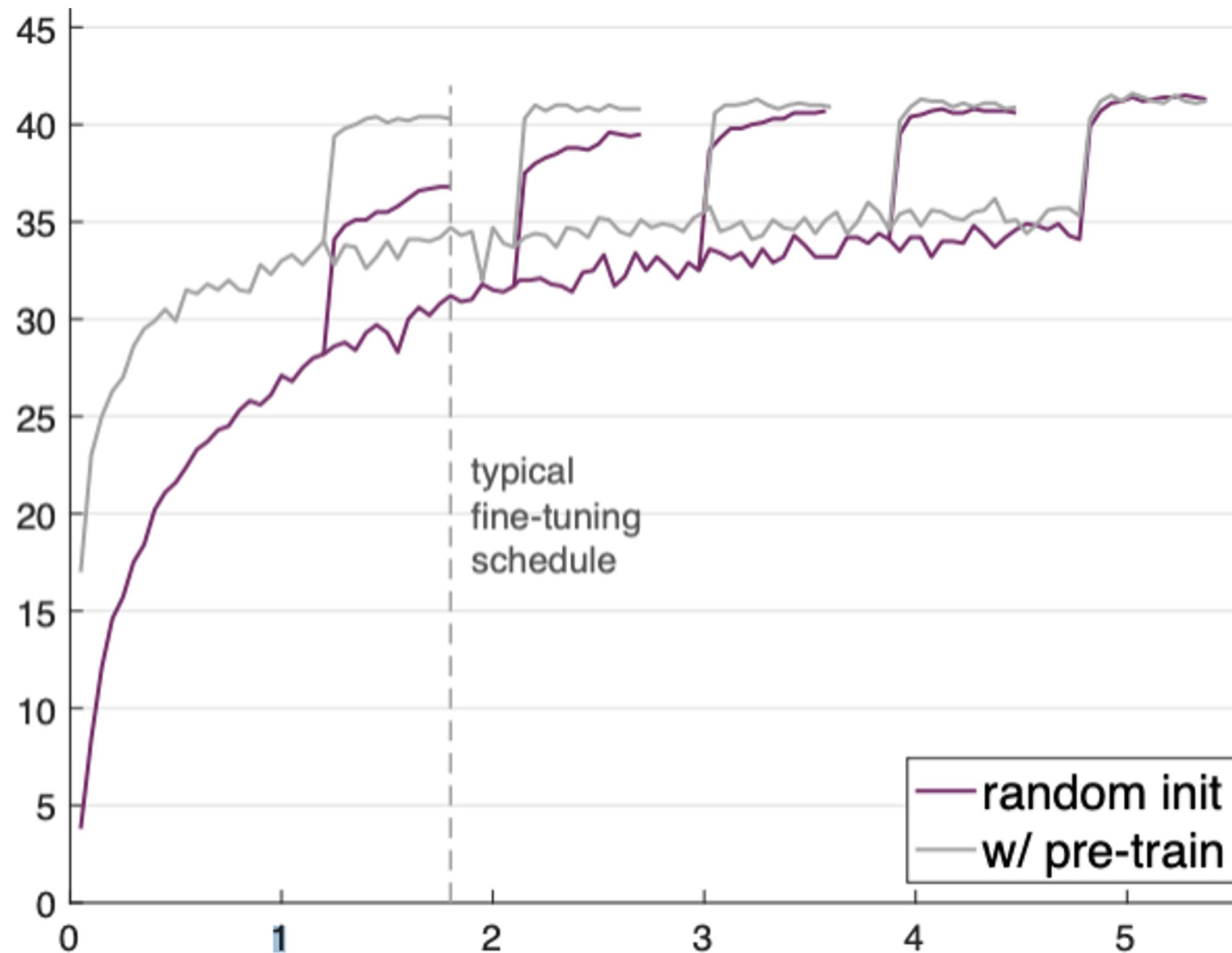
1. Train CNN on ImageNet
2. Fine-Tune (1) for object detection on Visual Genome
3. Train BERT language model on lots of text
4. Combine (2) and (3), train for joint image / language modeling
5. Fine-tune (5) for image captioning, visual question answering, etc.

Zhou et al, “Unified Vision-Language Pre-Training for Image Captioning and VQA”, arXiv 2019

Transfer Learning is pervasive!

Some very recent results have questioned it

COCO object detection



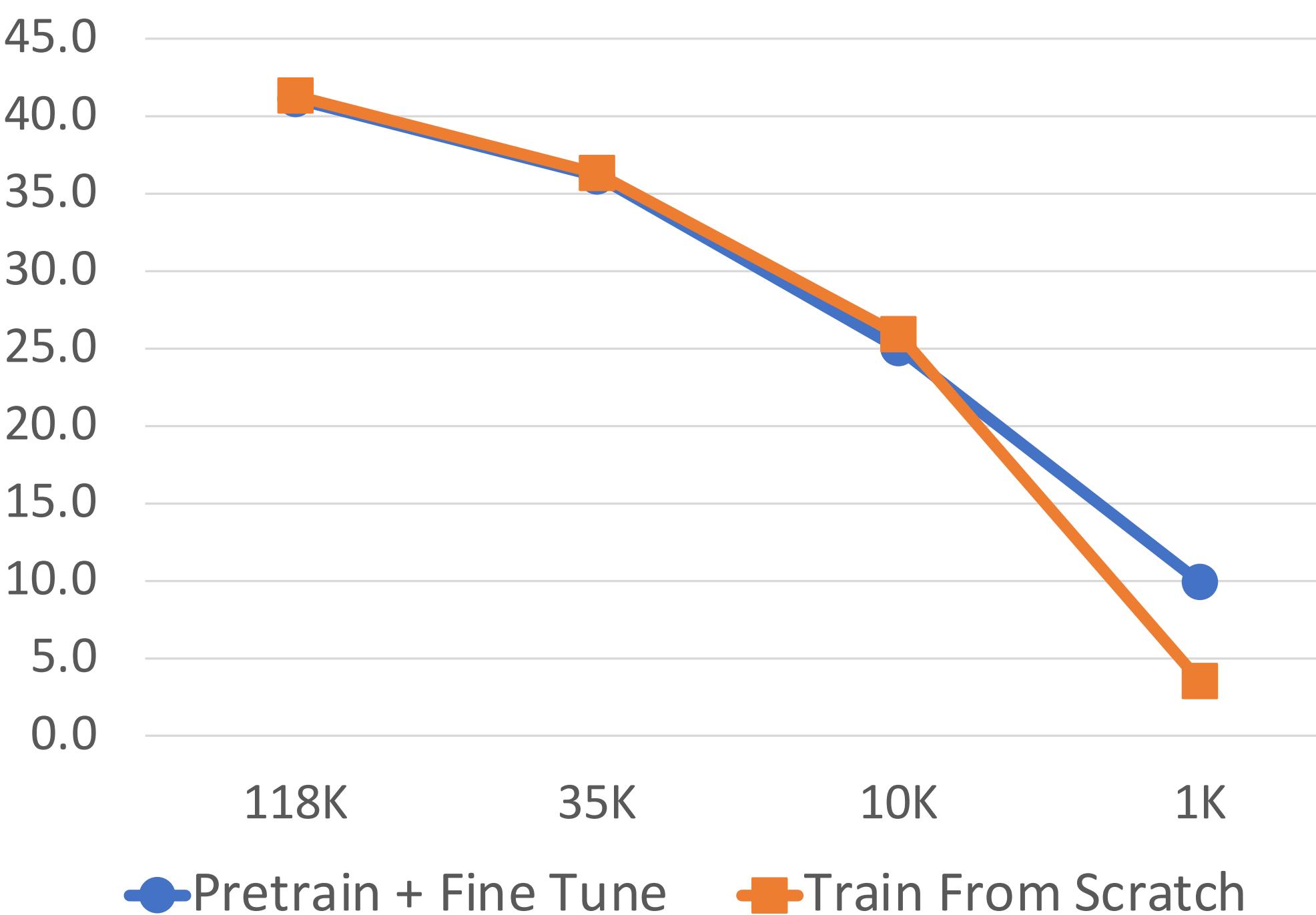
Training from scratch can work as well as
pertaining on ImageNet!

... if you train for 3x as long

Transfer Learning is pervasive!

Some very recent results have questioned it

COCO object detection



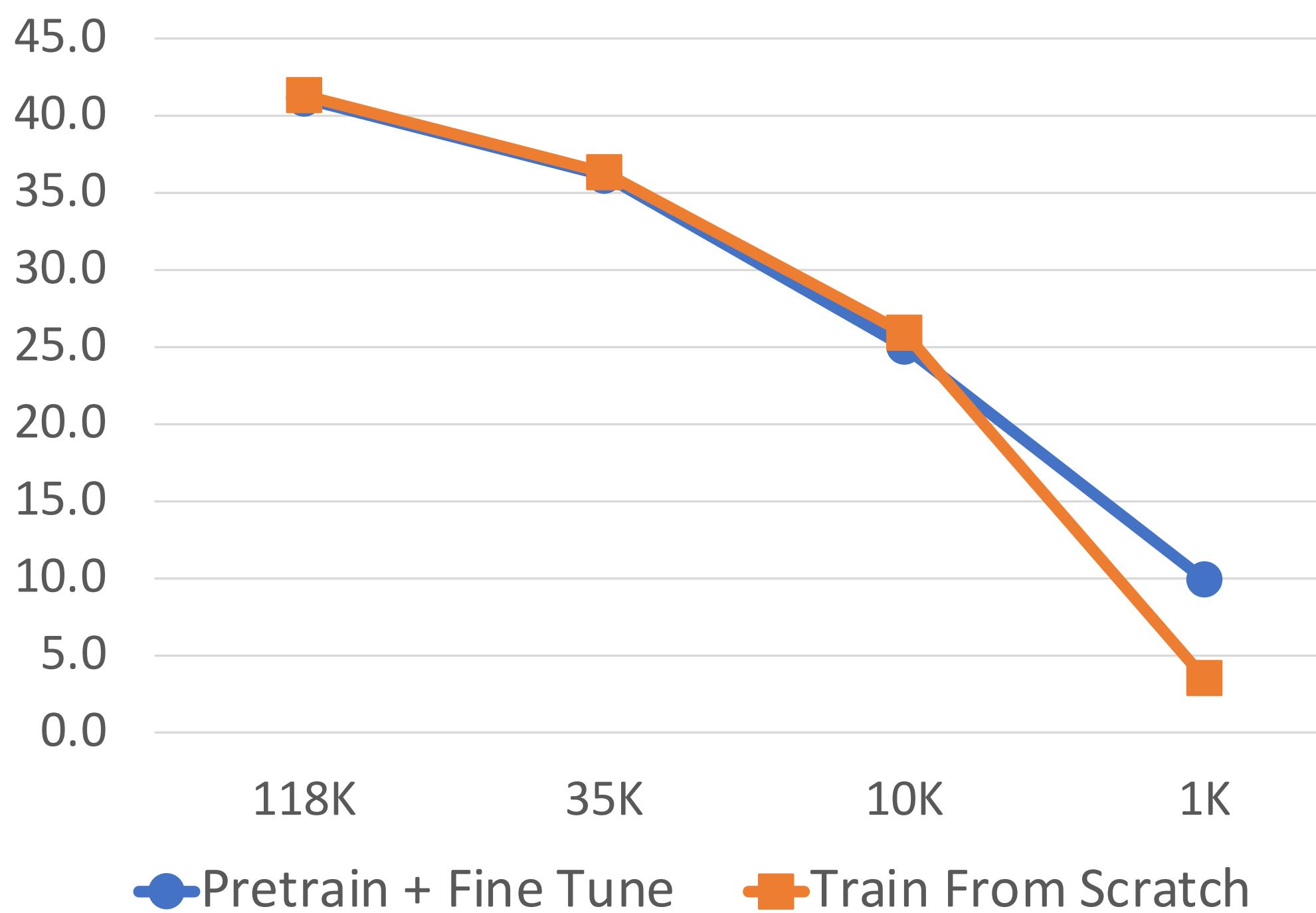
Pretraining + Finetuning beats training from scratch when dataset size is very small

Collecting more data is more effective than pretraining

Transfer Learning is pervasive!

Some very recent results have questioned it

COCO object detection



My current view on transfer learning:

- Pretrain + finetune makes your training faster, so practically very useful
- Training from scratch works well once you have enough data
- Lots of work left to be done

Summary

1. One time setup:

- Activation functions, data preprocessing, weight initialization, regularization

Last time

2. Training dynamics:

- Learning rate schedules; hyperparameter optimization

Today

3. After training:

- Model ensembles, transfer learning





Next Time: Deep Learning Software





DeepRob

Lecture 10
Training Neural Networks II
University of Michigan and University of Minnesota

